Metalevel and Reflexive Extension in Mechanical Theorem Proving

Sean Matthews

Ph.D. thesis

University of Edinburgh

1993

I declare that this thesis has been composed by me, Sean Matthews, and that the work described is my own.

Thanks

There are many people to thank here.

Most importantly, Alan Smaill, who tolerated me when I was at work, Bob Walker, who tolerated me when I was not at work, and my parents, who tolerated me.

Irene Moyna.

At work and after work: Fausto Giunchiglia for his enthusiasm, which interested me in this and other things, David Basin for intelligence and criticism, Andrew Stevens for flamboyant conversation, and the rest of the Mathematical Reasoning Group.

Alex Simpson, for incisive and constructive criticism and conversations.

Don Sannella, who was responsible for my coming to Edinburgh.

The various people who have made life in Edinburgh and other places good during the time: Heiner Igel, Peter and Lisa Elmes-Weinstock, Brian Rigney, Francesca Wüthrich, Geraint Wiggins, David Coombes, Frazer Macrae, Peter Donnelly, Hans Lehmann, Richard Froggatt, my family, many others.

John Fairleigh, for setting a good example.

Judy Delin.

This work was financed in part by a postgraduate grant from the Department of Education, Northern Ireland.

Part of this work is described in the paper 'Experience with FS_0 as a Logical Framework', to appear in in the formal proceedings of the second Logical Frameworks Workshop, published by Cambridge University Press.

Abstract

In spite of many years of research into mechanical assistance for mathematics it is still much more difficult to construct a proof on a machine than on paper. Of course this is partly because, unlike a proof on paper, a machine checked proof must be formal in the strictest sense of that word, but it is also because usually the ways of going about building proofs on a machine are limited compared to what a mathematician is used to. This thesis looks at some possible extensions to the range of tools available on a machine that might lend a user more flexibility in proving theorems, complementing whatever is already available.

In particular, it examines what is possible in a framework theorem prover. Such a system, if it is configured to prove theorems in a particular logic T, must have a formal description of the proof theory of T written in the framework theory F of the system. So it should be possible to use whatever facilities are available in F not only to prove theorems of T, but also theorems about T that can then be used in their turn to aid the user in building theorems of T.

The thesis is divided into three parts. The first describes the theory FS_0, which has been suggested by Feferman as a candidate for a framework theory suitable for doing meta-theory. The second describes some experiments with FS_0, proving meta-theorems. The third describes an experiment in extending the theory PRA, declared in FS_0, with a reflection facility.

More precisely, in the second section three theories are formalised: propositional logic, sorted predicate logic, and the lambda calculus (with a deBruijn style binding). For the first two the deduction theorem and the prenex normal form theorem are respectively proven. For the third, a relational definition of beta-reduction is replaced with an explicit function.

In the third section, a method is proposed for avoiding the work involved in building a full Gödel style proof predicate for a theory. It is suggested that the language be extended with quotation and subsitution facilities directly, instead of providing them as definitional extensions. With this, it is possible to exploit an observation of Solovay's that the Löb derivability conditions are sufficient to capture the schematic behaviour of a proof predicate. Combining this with a reflection schema is enough to produce a non-conservative extension of PRA, and this is demonstrated by some experiments.

Introduction

This thesis is intended as a contribution to work in the area of mechanical theorem proving. Specifically, it presents some experiments in the use of two particular meta-theoretic techniques to extend a theorem prover: meta-level reasoning and reflection. As well as this, it also considers the practical usability of the theory FS_0, proposed by Feferman as a framework suitable for this sort of work.

§1 Background

The research area of mechanical proof checking and development systems is now several decades old. Recently this work has become interesting to people like engineers (and so economically important) because of the problem of complex systems: our machines are becoming more and more intricate, and are increasingly being used in situations where unexpected behaviour could kill someone. One way that has been proposed to help ensure the reliability of these systems is to apply methods developed originally in mathematical logic to their analysis and construction. Unfortunately the enormous amount of mathematical work that is needed to do this for even the simplest machine can overwhelm an unassisted analyst.Mechanical proof development systems

might be a solution to this problem, since a machine is much better at reliably keeping track of large amounts of information than a person. However we have not yet been able to build a machine that allows formal proofs to be done on a machine as easily as informal proofs are done on paper. This means that it has not been possible to make proper use of methods of formal development.

Thus investigations into how proof development systems might be extended to embrace more of the techniques that mathematicians normally use are important: if successful they bring closer the goal of a practical mechanical proof development system; otherwise they highlight and clarify the problems that have to be tackled.

§2 Meta-level reasoning

A common device used in presenting a formal proof of a theorem in a logic textbook is to abbreviate it by appealing to new rules not actually defined in the theory. So even in what is described as a formal proof there will be gaps in the places where these new rules are used. Instead of filling these gaps with formal derivation, appeal is made to a *meta-theoretic* result stating that it is possible to fill in the gaps; this avoids the tedium of having actually to do the work, while still convincing a reader that a full formal proof exists. Meta-theory is this different view of a theory that is possible: not only can proofs in a formal theory be built but also, and as easily, proofs *about* a theory, so that it is possible, for instance, to prove that a particular class of gap in a proof can always be filled with a correct derivation.

Unlike textbooks most proof development systems cannot build or exploit meta-level proofs, and the alternative, of allowing users to add new rules to the system without formal justification, is clearly not safe. For this reason proof development systems have in general to build proofs solely from the basic rules supplied, and this can make them slow and cumbersome. One way that has been used to help this problem is to provide a programming 'metalanguage' at the interface, allowing users to automate the more repetitive parts of proof building; the same steps still have to be taken but at least they are being taken by the machine, which is faster and more reliable at handling details.

But why should it not be possible to have a proof development system simply make use of formal meta-theory in the same way that a book can? It is not *a priori* clear that this is can be done: the objects used to represent mathematics

have particular properties that might make the work of reasoning about them so great that it is not worth the effort. If this were the case it would be unfortunate: metamathematics is as proper a part of mathematics as any other, so we would have to conclude that there are parts of mathematics that just cannot be practically mechanised; we would also have to conclude that formal analysis of some important sorts of system (for instance, language interpreters) that raise the same sorts of problems as meta-theory is not practical.

The experiments presented here try to give some indication of what is possible by building meta-theoretic proofs for three very different theories: a proof of the deduction theorem for a very simple presentation of propositional logic; a derivation of a normal form theorem for a sequent calculus presentation of sorted predicate logic; and finally the derivation of a replacement beta-reduction rule for an equational presentation of the pure lambda calculus.

§3 Reflection

Probably the most famous result in mathematical logic is Gödel's proof that in general it is only possible to prove formally a strict subset of the true sentences of any part of mathematics. He did this by showing how, given a formal theory that has certain basic properties, it is possible to to formalise a description of the theory in itself, and use this to build a 'self-referential' sentence that is recognisably both true and not a theorem. Gödel showed how formal theories are incomplete, but he also indicated, as a corollary, how a formal theory could be extended so as to be, if not complete, then at least less incomplete.

There is a large problem with using the approach that Gödel describes though: his mechanism for building self referential sentences uses a 'quotation mechanism' and a 'proof predicate'; and the exercise of constructing these is so complex that it is not practically formalisable. Since in essence the second incompleteness theorem is a formalisation of the first, attempts to provide a proper proof have meant that a lot of work has been done on trying to abstract the distinctive properties of the proof predicate away from any particular version as a set of derivability conditions. This work suggests the possibility of extending a theory with a proof predicate while avoiding the work of Gödel's original approach. This immediately raises several questions. First, is it enough to define a proof predicate in terms of the derivability conditions,

or must it actually be instantiated with some particular concrete definition before the non-conservativity results hold; and second, is an extended theory constructed this way 'interesting', i.e., even given that the extension allows new theorems to be proven, do those new theorems tell us anything useful, or do they have the same artificial character as what Gödel constructed?

The second set of experiments that are described here looks at these questions. An extension of the language of first order predicate logic that adds a quotation mechanism directly is described, and this is used to provide a proof predicate for theories defined in this language directly by appealing to Löb's derivability conditions. Primitive recursive arithmetic is defined in this system, and three example proofs are presented: proof of a course of values induction schema, a proof of the definedness of Ackermann's function, and finally, a proof that primitive recursive arithmetic extended in this way contains Peano arithmetic.

§4 The theory FS_0

In order to do formal meta-theory some way of formalising theories so that it is possible both to reason in them, and to reason about them is needed. There are at least two ways that this can be done: either for each theory construct an *ad hoc* axiomatisation of it, or else use some system designed especially for giving descriptions of formal theories (a *Framework*, in which particular theories can be defined. The second approach is used here, of a single framework, which is used to give formal descriptions of the theories mentioned above and prove the necessary meta-theorems about them. The particular framework used is FS_0; a conservative, second order, extension of primitive recursive arithmetic proposed by Feferman as suitable for exactly this. It has not previously, so far as I or Feferman [27] is aware, been implemented, so this thesis is able for the first time to discuss the issues that are raised by doing so, and evaluate its utility.

§5 Structure of thesis

The structure of the rest of this thesis in chapters is as follows.

Chapter 2, 'The idea of meta-level reasoning'

This chapter is a discussion of general issues involved in metatheoretic reasoning, how it might be done, what the advantages and disadvantages compared to other approaches might be.

Chapter 3, 'The framework theory FS_0'

This chapter describes in detail the theory FS_0, its implementation and how it works in practice. The tail end of the chapter then describes a formalisation in FS_0 of a presentation of propositional logic, and proves the deduction theorem for it.

Chapter 4, 'Declaring a language'

This chapter describes how to define more complicated languages and theories; a sequent calculus presentation of sorted first order logic is given as an example.

Chapter 5, 'Meta-level reasoning'

This chapter describes two further experiments in meta-theoretic reasoning. First the construction of a derived rule for the theory described in chapter 4, that reduces expressions to prenex normal form; secondly, the definition of an equational presentation of the untyped lambda calculus, and the replacement of the beta-reduction rule with a more efficient one.

Chapter 6, 'The idea of reflection'

This chapter surveys theoretical work that has been done since Gödel on the subject of meta-level reasoning and reflection, and considers some of the implications that this might have for mechanical theorem proving. It examines particularly the Löb derivability conditions.

Chapter 7, 'Implementing reflection'

This chapter describes how to extend the theory defined in chapter 4 with a quotation mechanism and how this can be used to construct reflexive extensions of a theory defined on top of it.

Chapter 8, 'Using reflection'

This chapter describes three experiments in using reflection in constructing proofs in primitive recursive arithmetic. First a course of values schema is proven; then it is shown that Ackermann's function is defined, and thirdly, it is shown that a particular reflexive extension includes Peano arithmetic.

Chapter 9, 'Related work'

This chapter discusses related work on framework theories, meta-theoretic extension, and reflection.

Chapter 10, 'Conclusions and further work'

This chapter, as its title suggests, discusses what has been learned, and directions for further work.

The Idea of Meta-level Reasoning

This chapter discusses in a rigorous, but informal, manner the notion of a proof development system, and how this can be refined and extended with a *meta-level*. It does this by developing a simple theory and meta-theory for propositional logic, and then looking at the drawbacks of the meta-theory. Using what is learned from this exploration, the notion of a *framework theory* in introduced and developed.

§1 Historical background

Mathematics has traditionally been an informal enterprise where arguments have been sketched out on paper, with the reader expected to be able to fill in the small gaps in the proof using his own intuitions. There is a belief though, dating back to the beginning of the Western mathematical tradition that informal mathematics-on-paper can be translated into formal derivation; after an initial set of axioms and rules has been agreed on there should no longer be any need to appeal to the intuition of the reader of the proof. Euclid, in 'The Elements', made what can be regarded as one of the first attempts to develop this idea of proof from axioms. More recently a very determined attempt was make by Whitehead and Russell in [80]. The Principia is

sometimes described as the definitive example of an unreadable work of genius: three volumes of formal proof presented as a demonstration of the truth of the logicist thesis. And genius was needed — the effort of doing so much formal mathematics by hand was nearly superhuman and almost broke Russell.

If building large scale formal proofs by hand stretched Bertrand Russell to the limit it is not surprising that few others have tried to emulate him. Nevertheless, formal mathematics is now interesting for a second time, and maybe practically possible for the first time, because of the invention of computers. Computer software is essentially mathematical in nature, and some argue that, if mathematical techniques can be applied to its development, it will be easier to retain intellectual control [34], [41]. But the only reason why it is possible even to contemplate the sort of volume of formal mathematics that this involves is that computers themselves are ideal raw material for building the tools that are needed to do it.

§2 What is a proof development system

A *proof development system* (PDS) is a machine for helping a person to do formal mathematics. It ensures that only proofs that are valid in the particular formal system can be built. Not only that, but it should make the work of building a proof as easy and intuitive as possible. This in effect means trying to make the work of doing formal mathematics with the PDS resemble as closely as possible the work of doing informal mathematics-on-paper, by trying to follow a user's intuitions about the semantics of what the formalism describes, rather than forcing him to think in terms of the atomic notions that the particular formalism provides.

To illustrate this idea of a PDS, this next section develops a very simple PDS for propositional logic. Such a system is divided into three parts: The particular formal theory that is used to describe propositional logic; the *proof checker*, which ensures that only valid proofs in the formal system are allowed; and the interface between the proof checker and the user.

2.1 What is a formal theory

Before starting, the question of what a formal theory actually is needs to be addressed. For the purposes of this thesis, the definition is as follows: A *theory* is a subset of the objects in the language (*formulae*) collectively called the *theorems*, which are defined to be the closure of a set of formulae (called *axioms*) under a set of *primitive rules*. A rule in its turn is a relation between finite sets of formulae and formulae. It is usually written

$$\frac{\Gamma_1, \ldots \Gamma_n}{\Gamma}$$

where, given that $\Gamma_1, \ldots, \Gamma_n$ (known as the *premises*) are in the theory, Γ (the *conclusion*) is also in the theory.

A further definition that is useful at this point is for an *extension* of a theory: if $T' \supset T$ then T' is an extension of T.

It is important to distinguish primitive rules (or rules of *inference*) from *derived* and *admissible* rules, neither of which is primitive. A rule is said to be admissible in a theory if, whenever the premises are theorems of the theory, then so is the conclusion. A rule is derived if, for any extension with the same language and primitive rules, it is admissible[77].

2.2 A theory of propositional logic

The presentation of propositional logic given here is the propositional fragment of the presentation of predicate logic found in §2.6 of [68], which will be called SP. It is typical of the sort of rigorous presentation that is found in logic textbooks.

First, a notational convention is defined: $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ are variables over formulae of $\mathcal{L}_{SP}$. Then the formulae of the language $\mathcal{L}_{SP}$ of SP are defined by inductive definition as follows:

- The set of atomic propositions (not defined further here) is contained in $\mathcal{L}_{SP}$.
- If $\boldsymbol{A}$ is a proposition in $\mathcal{L}_{SP}$, then $\neg\boldsymbol{A}$ is a proposition in $\mathcal{L}_{SP}$.
- If $\boldsymbol{A}, \boldsymbol{B}$ are propositions in $\mathcal{L}_{SP}$, then $\boldsymbol{A} \vee \boldsymbol{B}$ is a proposition in $\mathcal{L}_{SP}$.

The theory SP is then defined as follows. The axioms are just all the formulae that are instances of the excluded middle,

$$\neg\boldsymbol{A} \vee \boldsymbol{A}. \qquad \textit{exmid}$$

And the rules of inference are:

$$\frac{A}{B \vee A} \quad \textit{expand}$$

$$\frac{A \vee A}{A} \quad \textit{contract}$$

$$\frac{A \vee (B \vee C)}{(A \vee B) \vee C} \quad \textit{associate}$$

$$\frac{A \vee B \qquad \neg A \vee C}{B \vee C} \quad \textit{cut}$$

2.3 A proof construction program

With the description of SP above it is possible to build a module of program code that can be used to construct proofs (i.e., a collection of functions that implement the necessary checking and data structures and allow proofs of particular propositions to be built). The code module will be called `PSP`, and the interface (the description of what parts of the proof constructor that other parts of a program that uses it can see) might look like:

```
module PSP is
    sorts :: wff,
             proof;
    functions :: mk_wff : string → wff,
                 neg : wff → wff,
                 or : wff → wff → wff,
                 left : wff → wff,
                 right : wff → wff,
                 interior : wff → wff,
                 goal : proof → wff,
                 exmid : wff → proof,
                 expand : wff → proof → proof,
                 contract : proof → proof,
                 associate : proof → proof,
                 cut: proof → proof → proof
end module
```

where the functions are programmed to behave as their names and types suggest. `wff` is the sort of well formed propositions and atomic propositions are constructed from arbitrary strings using `mk_wff`. Then new formulae can be built using `neg` and `or`, and taken apart again with `left`, `right` and `interior`. The last three are projection functions which, when applied to an appropriate well formed formula, return the left or right part of a disjunction, or the interior of a negation (remember that this is a module of *program code* though, so there is no guarantee apart from the programmer's efforts that a projection function will not be applied to the wrong sort of formula; if this happens then the code will have to indicate somehow that this is not permitted). Then proofs can be built in the obvious way using the functions named after the rules, and a proposition shown to be provable by some instance of `proof` can be recovered using `goal`.

It is only fair to point out that PSP is a particularly primitive implementation of *SP*, since, for instance, proofs have to be constructed 'bottom up' (i.e., the user cannot start with the proposition he wants to prove and reduce it stepwise — from the trunk out to the leaves — into smaller and smaller goals, but instead has to build a proof from the smallest proofs upward — from the leaves in to the trunk).

2.4 The user interface

With the proof checker built, the *user interface* can be constructed. This is the (probably much larger) part of the machine, that provides all the facilities that the user needs, so that constructing proofs in the proof checker is as easy as possible (or for some proof checkers, possible at all). Arbitrary facilities can be added to this according to whatever is needed, and whatever will help a user working in the theory. The user interface can consist of all sorts of things, but so long as, whatever else it does, it is forced to go through the module interface of the proof checker module, there is never any risk of building an incorrect proof.

One particular facility, common in PDSs that are used for doing real mathematics[i], is what is known as a *tactic language*. This can be a proper (usually interactive) programming language such as *ML*, *Lisp*, or *Prolog* which has been extended with the proof checker module. This means that, instead of a user having to suffer the demanding tedium of typing in each step of a proof by hand, he can instead write what are known as *tactics*: programs that automate the operations of well understood,

and often repeated proof steps. Such a facility of automating a lot of the proof construction can produce a difference in scale so great that it becomes a difference in kind, making possible formal proofs that it would simply not be humanly possible to build unassisted. For instance in [7] a proof of a version of Ramsey's theorem, which consisted of 64 steps entered at the interface, expanded to 17,531 primitive steps, which is not a practical proposition for manual entry.

When using tactics, the same steps have to be taken, but they can be performed as fast as the computer can do them, rather than as fast as the user can type them in. This is usually faster[ii], and certainly allows the user to think in a more abstract way, instead of at the level of single rule applications.

There are also ways in which a tactic language offers new methods of proof that are different by virtue of the amount of 'brute force' that a computer can apply to the problem. So for instance a problem might yield fairly easily to analysis by cases, but the number of cases would simply overwhelm a mathematician working with paper[iii].

Consider the simple example of the commutativity of $\vee$. Imagine if a user of PSP finds himself often having to prove propositions of the form $\boldsymbol{B} \vee \boldsymbol{A}$, given only that $\boldsymbol{A} \vee \boldsymbol{B}$ is provable. It would be possible, each time, to provide a proof by typing in something corresponding to the derivation:

$$\frac{\boldsymbol{A} \vee \boldsymbol{B} \qquad \neg \boldsymbol{A} \vee \boldsymbol{A}}{\boldsymbol{B} \vee \boldsymbol{A}}\ cut$$

(where the proof on the right is an instance of *exmid*). But this is time consuming, and means that the user has to be constantly aware of the individual operations that SP allows. A better approach is, instead, to write a program in the tactic language supplied by the interface, of the form:

```
def commute_or F = cut F exmid(left F));
```

which could then be used to commute the goal of any proof, without the user having to be concerned with the steps that are actually taken.

Another feature often found in PDSs is what is called a *lemma facility*. This is needed in systems that are designed to concentrate on one particular derivation at any time. Other derived formulae can be stored in a library or archive, and, if one matches a goal that has to be proven, then instead of building a new proof, it is enough to appeal to the fact that the goal has already been demonstrated to be

provable. This is a slightly narrower definition of a lemma than is usually found in mathematics-on-paper, where lemmas are not always restricted to being individual theorems in the theory. The commutativity of $\vee$ is a good example of the difference, in fact. In [68] it is presented in §3.1 as a *lemma*. But it is not a lemma in the sense here. It is stated as:

'if $\boldsymbol{A} \vee \boldsymbol{B}$ is provable, then $\boldsymbol{B} \vee \boldsymbol{A}$ is provable.'

But this is a relation between two derived results, not a particular derived result, and it is schematic. It is certainly true: the tactic `commute_or`, is, in one view, a proof of it, but this proof is not accessible to the PDS. There, lemmas are simply particular provable statements in the logic; there is no way to capture, once and for all, the fact that if it is possible to derive $\boldsymbol{A} \vee \boldsymbol{B}$ (for *any* disjuncts — not just particular instances) then it is possible to derive $\boldsymbol{B} \vee \boldsymbol{A}$. Instead, every time the result is needed, the tactic has to be run.

It has to be said that using propositional logic as an example here, while good for illustrating the limitation of a lemma facility, perhaps overstates that limitation. For instance a foundational higher order system which blurs the distinction between propositions and functions, such as intuitionistic type theory [57], will be able to exploit a lemma facility much more effectively than a first order system such as the specification language *Z* [41]. In the latter system obviously useful results like, e.g., special induction schemas, may not be provable as theorems (only every instance is). Instead, tactics have to be constructed to emulate these schemas, and these tactics, apart from taking time to execute, are often subtle in construction. In a higher order system, on the other hand, since it is possible to quantify over predicates, it is possible to prove as a theorem what in a first order theory is 'only' a schema.

§3 A more formal notion of a proof development system

The above outline presented a proof checker as a module of program code. But the behaviour of this module could also be presented formally, in a way that would amount to a theory of the theory of propositional logic. In fact, there are many ways that PSP could be formalised depending on what the formalism is for. A theory formalising PSP is a *meta-theory* of SP, and can be used to prove theorems *about* SP. SP in turn is called the *object-theory*. In what follows, a series of different meta-theories (denoted $\boldsymbol{MSP}$) of SP, which try to capture different parts of the theory of SP formally, will be discussed.

3.1 Preliminary remarks: provability and proof

The implementation of SP in PSP described above is able to show that formulae in the propositional calculus are provable, but it is not required to keep track of the details of how a formula was actually proved. No function is available in PSP that allows a user to find out how a formula was derived; all that he can know is that it was, by some valid means, demonstrated to be provable. This means that `proof` corresponds to what is known as a *provability predicate*. It would though, be easy to extend the PSP module with functions that returned information about the top level the proof. If this was done it would be possible to recursively extract the details of the derivation. This version of `proof`, retaining details of the proof, corresponds to what is known as a *proof predicate*.

A sensible question then is why a system needs to know how a formula is derived, as opposed to that it is derivable? After all, there are certainly PDSs (*Isabelle*, for instance) that do not keep track of the derivation (that is, they do not keep track of the primitive rules that go to make up the proof — this is not the same as keeping track of the instructions that the user gives to the PDS to construct the proof). On the other hand, there are systems that do keep track of the derivation, and make a good case for it; e.g., the *Nuprl* system, an implementation of a version of one of Martin-Löf's type theories. It not only implements the formal theory but also keeps track of how the derivations are constructed, so that the programs corresponding to the proofs can be constructed. This is not only in the spirit of Martin-Löf's own proposals, but also means that *Nuprl* can be used as a programming system. The same technique could be used, with an appropriate logic, for instance to synthesize

imperative programs, or electronic circuits [7].

Another, related, reason for keeping track of proof structure is that it is possible to construct tactics that perform transformations on the structure of the proof itself. Consider when a PDS is being used, as is suggested above for *Nuprl*, as a program synthesis facility; the witness (that is, the program corresponding to the proof) for the simplest proof of a goal may not correspond to a particularly efficient function. But it may be possible to construct tactics that can convert the proof automatically into a less intuitive one with a more efficient witness function [55]. A tactic that accesses the structure of a derivation is called a *transformation tactic* [19]. Unfortunately, a PDS that keeps track of the proof introduces complications into the relationship between the meta-theory and the object-theory in a PDS. This problem is discussed below.

3.2 A simple meta-theory of propositional logic

PSP is an *implementation* of SP in a programming language, it is not a theory. In this it is like most PDSs that are currently in use. Now consider what a meta-theory of SP would be like; this is what is presented next. It should be pointed out that the purpose of the developed example theory is to illustrate points, rather than to be paradigmatic of what might be used in practice, so it will not necessarily scale up directly to such. However, the differences between the example and what might be used in a real application do not affect the validity of the points made.

In the language $\mathcal{L}_{\boldsymbol{MSP^-}}$ of a meta-theory $\boldsymbol{MSP^-}$ of SP, There are two sorts of terms: those standing for formulae of $\mathcal{L}_{SP}$, and those standing for proofs in SP. In what follows $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ vary over $\mathcal{L}_{SP}$-sorted terms of $\boldsymbol{MSP^-}$, $\boldsymbol{D}$ and $\boldsymbol{D'}$ vary over 'proof-in-SP'-sorted terms (denoted $\mathcal{L}^d_{\boldsymbol{MSP^-}}$), and $\boldsymbol{P}$ and $\boldsymbol{P'}$ vary over formulae in $\mathcal{L}_{\boldsymbol{MSP^-}}$.

- $\langle\rangle$ is in $\mathcal{L}^d_{\boldsymbol{MSP^-}}$.
- If $\boldsymbol{A}$ is in $\mathcal{L}_{SP}$, and $\boldsymbol{D}, \boldsymbol{D'}$ are in $\mathcal{L}^d_{\boldsymbol{MSP^-}}$, then $\langle \boldsymbol{D}, \boldsymbol{A} \rangle$ and $\langle \boldsymbol{D}, \boldsymbol{D'}, \boldsymbol{A} \rangle$ are in $\mathcal{L}^d_{\boldsymbol{MSP^-}}$.
- If $\boldsymbol{A}, \boldsymbol{B}$ are in $\mathcal{L}_{SP}$, and $\boldsymbol{D}, \boldsymbol{D'}$ are in $\mathcal{L}^d_{\boldsymbol{MSP^-}}$, then $goal(\langle \boldsymbol{D}, \boldsymbol{A} \rangle, \boldsymbol{B})$ and $goal(\langle \boldsymbol{D}, \boldsymbol{D'}, \boldsymbol{A} \rangle, \boldsymbol{B})$ are in $\mathcal{L}_{\boldsymbol{MSP^-}}$.
- If $\boldsymbol{P}$ is in $\mathcal{L}^d_{\boldsymbol{MSP^-}}$ then $pr(\boldsymbol{P})$ is in $\mathcal{L}_{\boldsymbol{MSP^-}}$.
- If $\boldsymbol{P}, \boldsymbol{P'}$ are in $\mathcal{L}^d_{\boldsymbol{MSP^-}}$ then $\boldsymbol{P} \rightarrow \boldsymbol{P'}$ is in $\mathcal{L}_{\boldsymbol{MSP^-}}$.

Then the theory $\boldsymbol{MSP^-}$ is as follows. There are the two predicates $pr(\cdot)$ and $goal(\cdot,\cdot)$. The axiom schemas for pr are

$$pr(\langle\langle\rangle, \neg \boldsymbol{A} \vee \boldsymbol{A}\rangle) \qquad \textit{exmid}$$

$$pr(\langle \boldsymbol{D}, \boldsymbol{A} \vee \boldsymbol{A}\rangle) \rightarrow pr(\langle\langle \boldsymbol{D}, \boldsymbol{A} \vee \boldsymbol{A}\rangle, \boldsymbol{A}\rangle) \qquad \textit{contract}$$

$$pr(\langle \boldsymbol{D}, \boldsymbol{A}\rangle) \rightarrow pr(\langle\langle \boldsymbol{D}, \boldsymbol{A}\rangle, \boldsymbol{B} \vee \boldsymbol{A}\rangle) \qquad \textit{expand}$$

$$pr(\langle \boldsymbol{D}, \boldsymbol{A} \vee (\boldsymbol{B} \vee \boldsymbol{C})\rangle) \rightarrow pr(\langle\langle \boldsymbol{D}, \boldsymbol{A} \vee (\boldsymbol{B} \vee \boldsymbol{C})\rangle, (\boldsymbol{A} \vee \boldsymbol{B}) \vee \boldsymbol{C}\rangle) \qquad \textit{associate}$$

$$pr(\langle \boldsymbol{D}, \boldsymbol{A} \vee \boldsymbol{B}\rangle) \rightarrow pr(\langle \boldsymbol{D}', \neg \boldsymbol{A} \vee \boldsymbol{C}\rangle) \rightarrow pr(\langle\langle \boldsymbol{D}, \boldsymbol{A} \vee \boldsymbol{B}\rangle, \langle \boldsymbol{D}', \neg \boldsymbol{A} \vee \boldsymbol{C}\rangle, \boldsymbol{B} \vee \boldsymbol{C}\rangle) \qquad \textit{cut}$$

and for $goal$,

$$pr(\langle \boldsymbol{D}, \boldsymbol{D}', \boldsymbol{A}\rangle) \rightarrow goal(\langle \boldsymbol{D}, \boldsymbol{D}', \boldsymbol{A}\rangle, \boldsymbol{A})$$
$$pr(\langle \boldsymbol{D}, \boldsymbol{A}\rangle) \rightarrow goal(\langle \boldsymbol{D}, \boldsymbol{A}\rangle, \boldsymbol{A}) \qquad \textit{goal}$$

and, finally, there is one rule

$$\frac{\boldsymbol{A} \qquad \boldsymbol{A} \rightarrow \boldsymbol{B}}{\boldsymbol{B}}. \qquad \textit{detachment}$$

(where $\rightarrow$ is right associative. Notice that $\rightarrow$ here should not really be taken as being implication, since it is too weak, all that it is supposed to give is a theory that corresponds to the module PSP, i.e., a 'procedural' interpretation.) It is not hard to see that this theory fulfils the basic requirements of *adequacy* and *faithfulness*. (Adequacy is the property that if a proposition $\boldsymbol{A}$ is provable in the original theory SP, then it is possible using this theory to prove, for some $\boldsymbol{D}$, that $goal(\boldsymbol{D}, \boldsymbol{A})$. Faithfulness is the reverse of this, demanding that if it is possible to show, using the framework, that the encoding of a proposition $\boldsymbol{A}$ is provable, then there really is a proof in the original theory that $\boldsymbol{D}$).

Here a proof predicate formulation (if only implicitly) of the meta-theory is being used, since the structure of the derivation is carried around along with the derived statement in pr. It is not very useful as yet though, since there is no way of accessing that information in the theory; that can be done only from in the meta-level, with the perspective it provides. Extending the theory with a predicate that gives access to the theory is easy though: e.g., add a new predicate $subproof(\cdot,\cdot)$ as follows. The language $\mathcal{L}_{\boldsymbol{MSP^-}}$ is extended as follows:

- If $\boldsymbol{D}, \boldsymbol{D}'$ are in $\mathcal{L}^d_{\boldsymbol{MSP^-}}$, then $subproof(\boldsymbol{D}, \boldsymbol{D}')$ is in $\mathcal{L}_{\boldsymbol{MSP^-}}$.

and the theory $\boldsymbol{MSP}^-$ is extended with

$$pr(\langle \boldsymbol{D}, \boldsymbol{A} \rangle) \rightarrow subproof(\langle \boldsymbol{D}, \boldsymbol{A} \rangle, \boldsymbol{D})$$
$$pr(\langle \boldsymbol{D}, \boldsymbol{D}', \boldsymbol{A} \rangle) \rightarrow subproof(\langle \boldsymbol{D}, \boldsymbol{D}', \boldsymbol{A} \rangle, \boldsymbol{D})$$
$$pr(\langle \boldsymbol{D}, \boldsymbol{D}', \boldsymbol{A} \rangle) \rightarrow subproof(\langle \boldsymbol{D}, \boldsymbol{D}', \boldsymbol{A} \rangle, \boldsymbol{D}')$$

Modifying the theory in the other direction instead, so that is uses provability (call this $\boldsymbol{MSP}_N$), is simply a matter of defining a predicate $pr'(\cdot)$ which is like $pr(\cdot)$ except that it does not keep track of the derivation at all. This means that, for instance the axiom defining the provability of instances of the excluded middle would be simply

$$pr'(\boldsymbol{A} \vee \neg \boldsymbol{A}),$$

and with corresponding modifications for the other axioms (this means of course that $goal(\cdot, \cdot)$ is no longer needed at all, since $pr'(\cdot)$ carries around the goal, instead of the derivation). Below I will take $\boldsymbol{MSP}^-$ extended with *subproof* as the default, (though, as necessary, I will point out differences that arise with a provability predicate).

3.3 Working in the meta theory $\boldsymbol{MSP}^-$

In the presentation above, $\boldsymbol{MSP}^-$ is carefully designed to ensure that there is an isomorphism between proofs constructed in it, and proofs constructed in PSP.

For instance, it is possible to prove an instance of the commutativity of $\vee$ as follows: given that $\boldsymbol{A}, \boldsymbol{B}$ are propositions, and $\boldsymbol{D} = \langle \boldsymbol{D}', \boldsymbol{A} \vee \boldsymbol{B} \rangle$ is a derivation where $pr(\boldsymbol{D})$ then a proof that there is some $\boldsymbol{D}''$ such that $pr(\langle \boldsymbol{D}'', \boldsymbol{B} \vee \boldsymbol{A} \rangle)$ can be constructed as follows

$$\dfrac{pr(\langle \rangle, \neg \boldsymbol{A} \vee \boldsymbol{A}) \qquad \dfrac{pr(\boldsymbol{D}) \quad pr(\boldsymbol{D}) \rightarrow pr(\langle \langle \rangle, \neg \boldsymbol{A} \vee \boldsymbol{A} \rangle) \rightarrow pr(\langle \boldsymbol{D}, \langle \langle \rangle, \neg \boldsymbol{A} \vee \boldsymbol{A} \rangle, \neg \boldsymbol{B} \vee \boldsymbol{A} \rangle)}{pr(\langle \langle \rangle, \neg \boldsymbol{A} \vee \boldsymbol{A} \rangle) \rightarrow pr(\langle \boldsymbol{D}, \langle \langle \rangle, \neg \boldsymbol{A} \vee \boldsymbol{A} \rangle, \neg \boldsymbol{B} \vee \boldsymbol{A} \rangle)}}{pr(\langle \boldsymbol{D}, \langle \langle \rangle, \neg \boldsymbol{A} \vee \boldsymbol{A} \rangle, \boldsymbol{B} \vee \boldsymbol{A} \rangle)}$$

If this proof is examined, it is possible to see that it follows exactly the same path as `commute_or` defined above, with each application of *detachment* in the proof corresponding to a function application in the tactic. Unfortunately, this correspondence results in a weakest possible meta-theory; for instance not even the concept of a variable is available, so it is not possible to construct a proof of the commutativity

of $\vee$, which is quantified over the class of propositions, that can be instantiated later to the propositions that are actually needed in any particular case.

Since the theory $\boldsymbol{MSP}^-$ is, itself, a formal theory, it can be implemented on a machine. Why not implement a proof checker for it instead of building PSP, since any proof of SP can be reconstructed in $\boldsymbol{MSP}^-$? On the other hand, why bother? After all, implementing $\boldsymbol{MSP}^-$ is going to be more difficult than implementing SP, the resulting system will probably be slower, and it will not be able to do anything that PSP cannot do. But, while $\boldsymbol{MSP}^-$ cannot do anything that PSP cannot do, it is easy to extend it to a theory that can.

3.4 Adding variables to the meta-theory

$\boldsymbol{MSP}^-$ is a meta-theory of SP, but it is very weak. While all the theorems that it proves are 'meta-theorems' they are only meta-theorems in the most literal sense of the word — they do not say anything *general* about the theory SP, only about particular propositions in SP. In other words, it is possible to prove every instance of the commutativity lemma, but it is not possible to prove a statement of the form 'for any formulae $\boldsymbol{A}$ and $\boldsymbol{B}$, ...', which subsumes all those particular instances. In order to get this generalisation facility, the meta-theory needs to be extended at least with variables. The question of how to do this sensibly has to be addressed.

The theory $\boldsymbol{MSP}^-$ has only one connective, an arrow, and one rule, which allows the tail of a chain of arrows to be removed; the former suggests implication, and the latter modus ponens. However, while this interpretation may make sense, it is not captured in the theory — it is not possible to derive propositions that are provable for implication in SP.

Since the theory has arrows in it already, and the idea is to add variables, extending it to the the $\forall, \rightarrow$ fragment of predicate logic would seem to be sensible, so this can be taken as the first extension of $\boldsymbol{MSP}^-$; call it $\boldsymbol{MSP}^P$. The first thing that has to be done, is to extend the language to the language of $\boldsymbol{MSP}^P$, called $\mathcal{L}_{\boldsymbol{MSP}^P}$, with variables a, b, c, etc. over formulae in $\mathcal{L}_{SP}$, and extend $\mathcal{L}^d_{\boldsymbol{MSP}^-}$ with variables d, d', etc. In the following $\boldsymbol{P}, \boldsymbol{Q}$ vary over formulae in $\mathcal{L}_{\boldsymbol{MSP}^P}$, $\boldsymbol{v}$ varies over variables of $\mathcal{L}_{\boldsymbol{MSP}^P}$, and $\boldsymbol{t}$ varies over terms of $\mathcal{L}_{\boldsymbol{MSP}^P}$.

Then an axiom schema and rule for quantification over variables can be formu-

lated. The axiom is

$$\forall \boldsymbol{v}\, \boldsymbol{Q} \rightarrow \boldsymbol{Q}[\boldsymbol{t}/\boldsymbol{v}]$$

and the rule is

$$\frac{\boldsymbol{P} \rightarrow \boldsymbol{Q}}{\boldsymbol{P} \rightarrow \forall \boldsymbol{v}\, \boldsymbol{Q}}$$

(where $\boldsymbol{v}$ does not occur occur free in $\boldsymbol{P}$ and $[\boldsymbol{t}/\boldsymbol{v}]$ denotes the substitution of $\boldsymbol{t}$ for the variable $\boldsymbol{v}$ in the preceding formula in the usual way — i.e., avoiding variable capture and the like). Axiom schemas for $\rightarrow$ are

$$\boldsymbol{A} \rightarrow \boldsymbol{B} \rightarrow \boldsymbol{A}$$

$$(\boldsymbol{A} \rightarrow \boldsymbol{B}) \rightarrow (\boldsymbol{A} \rightarrow \boldsymbol{B} \rightarrow \boldsymbol{C}) \rightarrow \boldsymbol{A} \rightarrow \boldsymbol{C}$$

Given these extensions (and that the axioms are extended so that they can be used with variables), the theorem:

$$\vdash_{\boldsymbol{MSP}^P} \forall a \forall b \forall d\, (pr(\langle d, a \vee b\rangle) \rightarrow pr(\langle d, \langle\langle\rangle, \neg a \vee a\rangle, b \vee a\rangle)) \qquad (CL)$$

becomes provable. And since it is a simple theorem in the theory $\boldsymbol{MSP}^P$ it can be used as a lemma with the sort of lemma mechanism usually implemented in a PDS.

3.5 Further extensions

Even the theory $\boldsymbol{MSP}^P$ is a very small extension to $\boldsymbol{MSP}^-$, it adds only the fragment of predicate logic encompassing implication and universal quantification. It is easy to imagine much more powerful extensions, such as induction over derivations. But there are two problems that have to be taken into account: First, there is the risk that a proposed meta-theory will not be *conservative* with respect to the *pr* predicate. That is, an ill thought out extension $\boldsymbol{MSP}'$ of $\boldsymbol{MSP}^-$ might allow

$$\vdash_{\boldsymbol{MSP}'} pr(\langle \boldsymbol{D}, \boldsymbol{A}\rangle) \qquad \textit{even though not} \qquad \vdash_{\boldsymbol{MSP}^-} pr(\langle \boldsymbol{D}, \boldsymbol{A}\rangle)$$

(for some $\boldsymbol{D}$ and $\boldsymbol{A}$). On the other hand, there is no reason why, as long as the proposed extended theory is faithful, it should not be used. And the resources that such an extended theory could provide might make all sorts of useful results possible. Secondly, even if the extended theory is faithful, there is the separate problem that a lot of useful meta-level results will not be provable using any extension of this sort,

no matter how strong. For instance, in any introductory work on proof theory, one of the first meta-level theorems proven is the deduction theorem. This says simply that

$$\vdash_T \neg \boldsymbol{A} \vee \boldsymbol{B} \qquad \textit{iff} \qquad \vdash_{T[\boldsymbol{A}]} \boldsymbol{B}$$

(where $T[\boldsymbol{A}]$ means the theory T extended with the axiom $\boldsymbol{A}$, and $\vdash_T \boldsymbol{A}$ means $\boldsymbol{A}$ is provable in theory T). This is not provable in any obvious sensible extension of $\boldsymbol{MSP}^P$, since it states a relationship between two theories rather than a relationship between two proofs in the one theory. This is, in a way, the next step up from the problem illustrated by the commutativity lemma, that is common in PDSs used in practice. But even the commutativity lemma relates different derivations in the same theory, whereas even to state the the deduction theorem some way of dealing with the general notion of a theory is needed. This would tend to suggest that designing a meta-theory for a system 'bottom up', i.e., the way that a meta-theory for SP has just been developed, by *ad hoc* extensions from a simple initial theory, is doomed to fail, since there are results that need a fundamentally more general notion. The deduction theorem is not, either, an isolated result; an examination of any proof theory text will show that there are a lot of results like it that have obvious applications in computer assisted theorem proving (some of these are discussed later). Thus the next section looks at 'top down' approaches to the problem, i.e., systems that are designed to be able to define large classes of very different logics easily.

§4 Meta-theory and frameworks

The section above shows that it is not really possible to develop a general notion of meta-theory 'bottom up'. On the other hand it also suggests that a general facility for meta-theory might be useful. Logicians have in the past looked at this notion of a general meta-theory [64], [73]. More recently, computer scientists have coined the term *logical framework*, or framework, to describe it [20].

4.1 What are useful properties of an abstract theory?

The first question that has to be asked, before particular approaches to the idea of a framework can be examined, is 'what exactly does a good framework consist in?' Some indication of what is needed might be got by examining the deduction theorem, and imagining what a formal expression of it would look like. For example using the sort of formal language used above, it would look like

$$\forall d \forall a \forall b \forall \Gamma \, (pr(d, \Gamma \cup \{a\}) \wedge goal(d, b) \rightarrow \exists d' (pr(d', \Gamma) \wedge goal(d', \neg a \vee b))) \quad (DT)$$

(where Γ is a set of axioms, and $pr(\cdot_1, \cdot_2)$ is now a two place predicate which holds if and only if $\cdot_1$ is a derivation that is valid for SP extended with the set of axioms $\cdot_2$). But this is still only an extension to first order theories in general. It does not take account of more dramatically different theories such as, say, the theory of finite types (FT). And yet, later on in this chapter a result relating a first order theory with FT will be discussed. So extending the proof predicate to consider any first order theory has not really solved the problem.

What is really needed is a much more general meta-theory, that is able to treat very different theories and relate them together, and it seems that it is not going to be possible to develop such a thing using the approach followed so far, of making ad-hoc extensions to a simple axiomatisation of the original object theory.

Designing such a theory is not an easy task. For instance, on examination, the notion of rule application is much more complex than it appears from the examples above: propositional logic is very unusual in the simplicity of the rules that it uses. In theories that are used in practice (even like $\boldsymbol{MSP}^-$), rule applications can be arbitrarily complicated, and some way of dealing with this has to be provided. Substitution of terms for variables in formulae is the most obvious (though not the only) example of this.

The system should also be flexible in the way it allows proofs to be built. A system that insists that a proof be constructed bottom up is probably not going to be much use for practical proof construction. And, finally, but very importantly, since in the end a framework is intended for proving theorems that are equivalent to theorems in the object-theory, it is important that framework proofs do correspond to proofs in the object theory. (like, for example, above, where there is no real difference between proving an instance of the commutativity of $\vee$ in PSP, or in $\boldsymbol{MSP}^-$). At the same

time it should be possible to prove more general theorems in the framework itself, that do things like show interesting relationships between different object theories; thus a framework that only allowed meta-theories closely related to, for instance, $\boldsymbol{MSP}^-$ to be defined is not what is wanted.

4.2 Foundational theories

This is a good point at which to make the distinction between foundational theories and framework theories. Foundational theories are systems like Zermelo Frankel set theory with choice (ZFC), the theory of Principia Mathematica (PM) or Intuitionistic Type Theory (ITT), which are proposed as very general theories which in some sense are able to subsume any particular piece of real mathematics. For instance, it would be possible to do classical number theory in ZFC by constructing sets with the properties that we expect the numbers to have, and then working with those sets instead of with some formal system describing the integers directly.

A framework theory, on the other hand, does not do this, but rather provides a facility for describing the formal system itself, and manipulating that. This allows, for instance, a framework to prove theorems in an inconsistent formal theory (this may not be an advantage). A foundational theory, on the other hand, cannot do this, since only if the theory is consistent (assuming that the foundational theory itself is consistent) will it be possible to build objects in it that satisfy the properties that the formal theory stipulates[iv].

4.3 Post-style systems

There are two major styles of framework theory available in practice. The style that is more familiar to computer scientists is based on theories of the typed lambda calculus (which are discussed in the related work chapter), but there is another, older approach. This was originally developed by Post [64] in his attempts to provide a decision procedure for PM. This was based on recursively enumerable strings. Post's presentation was not really intended for the notion of framework theory presented here: for instance, it had no facilities for doing 'useful' metatheory, and using strings as the basic data structure means that it would be difficult to implement in a practical manner. But it contained the essential idea of presenting a theory as a recursively enumerable class of strings of symbols, and showed that even very large theories such as PM could be presented in this way, using only a very small collection of operators.

Recently, a new version of Post's approach to presenting a formal theory has been suggested by Feferman [25], in a form intendeded to be a practical, as much as a theoretical, tool. Feferman's theory uses s-expressions rather than strings, and is a conservative extension of PRA, which gives induction over Σ^0_1-classes (i.e., those that are recursively enumerable). In this version, a Post-style encoding of formal theories becomes a practical possibility, and a quite powerful generalisation facility is available to allow useful meta-level reasoning, unlike Post's suggestion. Feferman's system is discussed at length in the next chapter, but it is worth making some points about his system now.

For instance, one possible problem with the Post approach even in Feferman's version, is that operations such as substitution need to be coded up by the user instead of being something that he can count on automatically, like in a type theory. To counter this, though, the presentation is very close to the informal notion of what exactly a formal system is, and it is easy to add computational facilities that make constructing a definition of substitution (or anything else) quite easy.

§5 Using a framework

Having discussed the nature of a framework theory, the issues that meta-level results like the deduction theorem raise can be examined. For the sake of what follows, it will be assumed that $\mathcal{F}$ is a framework theory that allows theories to be declared and reasoned about in the way that has been discussed. The notation of $\boldsymbol{MSP}$ above will continue to be used but this should not be taken as anything more than a convenience.

5.1 Working in two theories

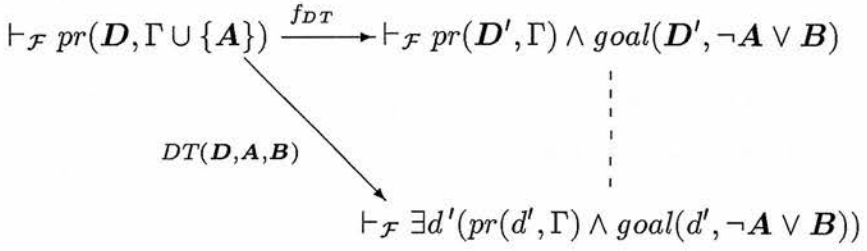
In a framework system, there are two, perhaps very different, systems to be dealt with: the framework theory itself, and the object theory (or theories) declared inside it. This means that it becomes necessary to distinguish between two different sorts of tactics: *framework tactics*, which are designed for manipulations of the framework logic itself, and *object tactics*, or again just tactics, which are designed for manipulations of the declared logic. The second of these is obviously a subset of the first, since derivations in the declared logic correspond directly to manipulations in the framework. An example of an object tactic would be the equivalent of `commute_or`

for the theory ***MSP*** (which would automate the steps described above, in §3.3, to commute a disjunction).

5.2 Using a meta-level result

In the commutativity lemma, where the logic used only the $\forall, \rightarrow$ fragment of a predicate logic, the lemma had to state the structure of the object-level proof in the consequent of the implication, and the proof of the lemma consisted in showing that the outline was instantiated in every instance. So whenever the commutativity lemma was applied, the proof was constructed as a side effect.

For (DT) in §4, the situation is different; all that is available from the theorem is a statement that some such object must exist. However, since the standard proof of DT is constructive, it is possible, by examining it, to build a framework tactic f_{DT} that will transform any proven instance $pr(\boldsymbol{D}, \Gamma \cup \{\boldsymbol{A}\})$, where $\boldsymbol{D}$ is a proof of $\boldsymbol{B}$ from the set of axioms $\Gamma \cup \{\boldsymbol{A}\}$, into an instance $pr(\boldsymbol{D'}, \Gamma)$, where $\boldsymbol{D'}$ is a proof of $\neg A \vee B$ from the set of axioms Γ. But this is not satisfactory — the reason for doing meta-theory in the first place is to avoid having to do this at all. The situation can be illustrated by a diagram as follows:



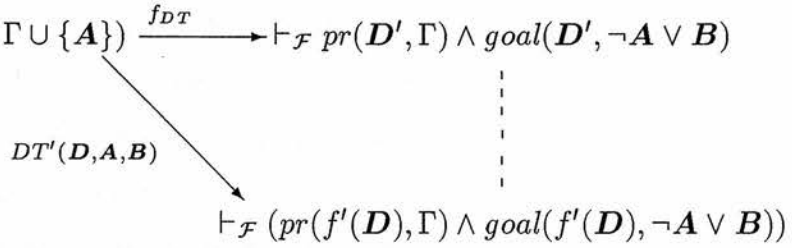
and the problem is with the dashed relation. All that DT does is to say that that there exists a proof that $\neg \boldsymbol{A} \vee \boldsymbol{B}$ from Γ, given only a proof that $\boldsymbol{B}$ follows from $\Gamma \cup \{\boldsymbol{A}\}$, it does not say what that proof is.

One possible way to deal with this problem is to add a facility to the framework that allows computable functions to be defined and executed. Then it might be possible to build an appropriate function f' inside the framework using the

facilities supplied by the logic (given that it is suitable), that satisfies the existential quantifier[v], i.e., for which it was possible to show

$$\forall d \forall a \forall b \forall \Gamma \, (pr(d, \Gamma \cup \{a\}) \wedge goal(d, b) \rightarrow pr(f'(d), \Gamma) \wedge goal(f'(d), \neg a \vee b)) \quad (DT')$$

If this is done instead, the diagram looks like



Since the usual proof of the deduction theorem is constructive, actually building f' is not difficult. The problem comes when the proof of a theorem is not constructive, and it may not be obvious how to build the function that instantiates the existential quantifier. Since it is questionable how useful a non-constructive proof of a meta-theorem would be, in the light of the need for some way to construct the derivation, why not restrict the meta-theory in the first place, so that it is always possible to retrieve a proof[vi]. With this restriction, the proof of the meta-theorem contains, implicitly, details of how to construct an object which can be used to instantiate the existential. Thus, with a constructive logic there is really no difference between DT and DT'.

The deduction theorem is a very simple example of a meta-level theorem, but it illustrates a common situation. Before moving on, it is worth pointing out that a much more general schema (which subsumes the deduction theorem) would be of the form

$$\forall d \forall \Gamma \, pr(d, \Gamma) \wedge \boldsymbol{P_1}(d, \Gamma) \rightarrow \exists d' \exists \Gamma' \left(\boldsymbol{P_2}(d, \Gamma, d', \Gamma') \wedge pr(d', \Gamma') \right) \quad (GMT)$$

here $\boldsymbol{P_1}$ is a predicate checking that the given proof is of a certain form, while $\boldsymbol{P_2}$ defines the relationship between the new and the old derivations. Notice that it is very similar to the pre- and post- conditions that are used in some styles of program verification.

5.3 Other implications

One other thing that that should be pointed out about DT above, is that, once it is proven, it is very broad in its application. It can be used with any theory that use the rules of SP. In fact if the result is proven, as Shoenfield does (his system also includes a rule for existential quantifiers), then the result can be used with any first order mathematical theory.

Also, remember that the deduction theorem was chosen because it was an example of a result that relates two theories together, rather than two results in the one theory. There is clearly no reason, in a framework that can prove the deduction theorem, why it should not be possible to prove relations between radically different theories, allowing a user constructing a formal proof to exploit the technique of mapping the theory into another, and showing that the result holds in isomorphic image, as a way of proving the goal. Again, this is discussed in the next section.

§6 Advantages that meta-level reasoning brings

Having discussed what is needed to do meta level reasoning, I now want to examine some of the practical advantages that it might bring to a system.

6.1 Complexity advantages

In discussing the example of the deduction theorem DT above, I considered how to construct the proof object that DT proved existed. But most of the time the user will not actually be interested in this. All he will be interested in is that the goal $\neg \boldsymbol{A} \vee \boldsymbol{B}$ has been reduced to $\boldsymbol{B}$, if he is allowed instead simply to assume $\boldsymbol{A}$ as an axiom. The structure of the proof is irrelevant, but it is building the proof that consumes the machine time. This is why a tactic can take so long: it guarantees that an invalid proof cannot be constructed by the simple, but time and resource consuming, method of constructing a proof from the primitive rules. On the other hand, moving the antecedent of the goal into the axiom set is a constant time operation. The former operation takes a large and, to the user, arbitrary amount of time while the second will always take the same very small amount of time.

6.2 Intellectual control

Another problem with a tactic, apart from speed (or slowness) is that the only information about its behaviour is intensional — no extensional abstraction is available. For instance an examination of the code of a tactic corresponding to the deduction theorem will reveal only a complicated function that describes a proof transformation. On the other hand, there will be no hint of the simple transformation that is the end result, and, without careful documentation the user is unlikely to be able to figure out what it does at all. Further, there is no way to guarantee that the code does what it is supposed to do, even when it is being built. And it is a truism that there are always bugs in software. The situation is relieved to some extent for tactics because they have to work through a proof checker, which means that a bug cannot result in an invalid proof, only in a failure to construct anything at all. But this is not a great help — if complicated code fails months after it was built, tracking down the error is difficult and time consuming work. With a meta-theorem, on the other hand, there is the same security as there is with verified program code, so there should be no such problem.

Further, as tactics are combined together in the way that they are supposed to be, it becomes more and more difficult to keep track of what exactly they are supposed to be doing. A meta-level facility means that, since there is a formal and explicit description of what a tactic does, it is much easier to manage the system.

6.3 The reliability of the proof checker

In the description of a typical PDS above I suggested that the proof checker part of the system is minuscule. While in the ideal this is true, in practice it is not. Usually, while a PDS is advertised as being an implementation of a logic that can be described on paper in a few pages, the logic that is actually programmed is much more complex, with the basic rules supplemented with all sorts of *decision procedures* and *normal-form procedures*. These new rules have a different character from the sorts of new rules that are sometimes introduced into a paper discussion about a theory. These latter are introduced in order to make an exposition clearer, whereas the former are intended to make the exposition faster; i.e., to make certain operations more efficient by appealing to syntactic information so that it is not necessary to construct the rule at all. In essence they are the informal equivalents of meta-

theorems.

There is another important difference between logics on machines and logics on paper: while the rules of a paper description are designed, usually, to be as comprehensible as possible, the design of rules such as decision procedures and the like, which supplement the usual rules in the machine, make no concessions to comprehensibility — they are supposed to be as efficient as possible. Unfortunately efficiency often requires intricacy, which in turn implies difficult to find bugs. There is, in short, no guarantee that these rules do not introduce inconsistency through an implementation error, and unlike with the rules of the paper presentation, simply inspecting the code is not likely to be a useful guarantee[vii]. It is ironic that the first thing that is often done to a PDS is to extend the logic it implements in ways that are not guaranteed to be conservative, or even sound.

With a meta-level facility though, this is no longer a problem, since it is no longer necessary, for the sake of speed, to build such facilities into the basic logic. Instead, they can be verified as meta level theorems and safely implemented at that level as lemmas (like, e.g., DT or CL).

6.4 More general reasoning facilities are possible

One particularly interesting facility that meta-level reasoning offers is in the way that very different theories can be related. For instance, a common interest of users of PDS systems is program verification or synthesis. Specifically interesting are proofs of Π^0_2-propositions (those of the form $\forall x\, \exists y\, P(x, y)$, where P is an open predicate defining the relationship between the input parameters x and the result y of a function application). Consider the following example.

It is well known that for constructive formal theories such as HA a proof asserting the existence of an object contains sufficient information so that such an object can be constructed given a suitable theory of recursive functions. So in an appropriate framework theory it should be possible to construct a meta-level proof showing this correspondence, so that functions in some (formalised) programming language, can be built by examining proofs in HA. HA, since it is easier to reason in, could then be thought of as a specification language/theory.

This would be an interesting relation in itself, but a further very useful step is

possible: it is also known [61] that for Π^0_2-propositions

$$\vdash_{PA} \forall x\, \exists y\, P(x,y) \qquad \textit{iff} \qquad \vdash_{HA} \forall x\, \exists y\, P(x,y).$$

Now, if this theorem is formalised properly at the meta-level, the development of a program can proceed as follows: First show, taking advantage of all the tools that can be used in a classical theory, that

$$\vdash_{PA} \forall x\, \exists y\, P(x,y).$$

Then, using results like those just described, the proof can be transformed into an equivalent in HA, and from that a function implementing the specification can be recovered[viii].

An examination of a good text in proof theory will show that there are plenty of other results like this that can be exploited to make a proof easier, and that correspond to the common informal mathematical device of establishing an isomorphism between an unsolved problem in one theory and a solved problem in another theory.

§7 Disadvantages of meta-theory

In the last section I argued for some of the possible advantages of meta-level reasoning as another facility for a PDS. To be fair, this has to be balanced with a discussion of some of the possible disadvantages that might be associated with it.

7.1 Difficulty of doing formal meta-theory

Meta-theory has a reputation for being difficult. The reason most often given for this is that bound variables are supposed to be extremely difficult to reason about neatly [67]; worries about variable capture constantly get in the way of the construction of proofs to such an extent as to render the effort required impractical. In attempts to get around this problem, new approaches have been tried that are designed especially for meta-theory, such as de Bruijn [11] indices, and these have eased the problem enough to get some things done. Shankar, for instance, first tried to used a standard binding scheme in his formal proof that the lambda calculus is Church-Rosser [67], but admitted defeat. He was eventually successful when he used a de Bruijn style binding mechanism. The problem with approaches like that which de Bruijn suggests, is that while they clarify the meta-theory, they obscure the object-theory. Working in, as

opposed to working with, a theory that uses this approach is practically impossible for people (though it is extremely easy for machines, with their different perspective).

I would not dispute that this problem with binding is, at least to an extent, a real difficulty. There are times when a tactic is much more useful than a meta-theorem. But it is possible to divide labour effectively between tactics and meta-theory, rather than have one or the other do the whole.

Consider (GMT) in §5.2. This can often be split up into two subtheorems:

$$\forall d \forall \Gamma \forall d' \forall \Gamma' pr(d,\Gamma) \wedge \boldsymbol{P}_1(d,\Gamma) \wedge \boldsymbol{P}_2(d,\Gamma,d',\Gamma') \rightarrow pr(d',\Gamma') \qquad (GMT_1)$$

and

$$\forall d \forall \Gamma\, pr(d,\Gamma) \wedge \boldsymbol{P}_1(d,\Gamma) \rightarrow \exists d' \exists \Gamma'\, \boldsymbol{P}_2(d,\Gamma,d',\Gamma') \qquad (GMT_2)$$

from which (GMT) follows. It is possible to dispose of one of these for all time with a meta-theorem, and the other, each time, with a tactic. The part that is easier to deal with using meta-theory is (GMT_1), since instead of having to instantiate an existential (which corresponds to building a function to perform the syntactic relation, and having to cope with the problems of avoiding capture etc.), it is only necessary to show that, given that the syntactic transformation is possible, there is a proof that supports it (which may be easier). Then a framework tactic, instead of a function in the framework theory itself, can be used to construct an appropriate b' for b, and then go on to prove that it satisfies the relation $\boldsymbol{P}_2(b,\Gamma,b',\Gamma')$ (a function in the framework would not need to do this, since the result of evaluating it would already be guaranteed to satisfy the relation). Since this is done informally I can hope that it will be much easier.

This achieves a satisfying balance between meta-theory and object-theory, and it is still not necessary to construct the proof which is where most of the complexity should lie — in fact the construction of a proof that satisfies the relation $\boldsymbol{P}(\cdot,\cdot)$ is much more likely to be of the same complexity as a function embedded fully in the language, even if with a much larger constant.

More generally though, the problem is just that proving a meta-theorem is more time consuming, and more demanding, than constructing a tactic. Especially if the tactic is only going to be used a few times, it will often just not make sense to devote the time to the work (unless the meta theorem gives an enormous complexity advantage).

7.2 Problems with the logic

One advantage that a tactic language has over even a constructive framework logic is that it is *Turing complete*, i.e., any algorithm can be expressed in it. This is usually not the case with program logics. As a formal system, a program logic has an upper bound on expressiveness, so there will be functions that can be programmed in the tactic language that cannot be programmed in the logic. This is not as bad as it sounds since, in practice, a function for any result that a user might want to compute can be expressed in a theory only as strong as PRA^{ix}. But, even if it is possible to construct the function, the way that a programming logic forces a user to express himself may be very awkward, and mean even that he has to use a less efficient algorithm than otherwise, negating the advantage of using meta-theory[x].

It may also be the case that the complexity of the transformation of the goal that corresponds to the meta-theorem is the same as that of the tactic it is expected to replace. Even here though, there may still be an improvement, since with a tactic a lot of side conditions have to be checked for each rule application that can be proven to hold for a function implementing the meta-theorem. With a verified function, on the other hand, none of this 'run time type checking' need be done — it will all have been checked when the function was being verified. So a meta-theorem, even of the same complexity, may have a worthwhile constant factor advantage.

On the other hand, it is also true to say that not every tactic is 'formal' in a way that allows it to be expressed and proven as a theorem in mathematical logic. The search tactics that would be needed to implement the sort of meta-level planning above, for instance, would be difficult to formalise, unless the framework had access to some concept of 'plausible'.

§8 Conclusions

This chapter has looked at what might be possible if a PDS was able to use meta-theory in order to extend the theories defined in it, and what a good approach to providing meta-theory might be. It was argued that meta-theory provides advantages such as simplicity, speed, control and flexibility. Against this it was also suggested that that some large problems might outweigh these advantages, making the enterprise impractical.

Also, various ways of actually providing a meta-theory facility were considered and assessed: The naïve 'bottom up' approach of directly axiomatising the object theory of the PDS was considered and rejected, because it was was not able to cope with important sorts of meta-theoretic results. After this, the notion of designing a meta theory 'top down' was considered instead. To this end the idea of a framework theory was introduced; i.e., a theory that captures a much more general idea of meta-theory. What the useful properties of such a theory might be was discussed.

Notes

i. A tactic language was not always so common — in the *AUTOMATH* system, which is the first where 'real' mathematics was done, there was no such facility. But *AUTOMATH* would no longer be considered to be a good platform for formal proof development. There are other systems that take an approach where a tactic language is not needed — e.g., the Boyer-Moore theorem proving system [17], or the Otter theorem proving system [53], where, instead, there is a fairly powerful uniform proof strategy.

ii. A tactic is not always faster than a user though — if the tactic tries to do something very complex, it may have to perform a lot of search, where a user can make use of experience to remove the need for much exploration. But even then, tactics still allow the user to think at a more abstract level.

iii. An example of a proof that uses an enormous case analysis is, for example, the first proof of the four colour problem; this consists of an exhaustive examination of several thousand different possible cases, to check that they all have a necessary property — it would be practically impossible for a person to carry out the same checks, or to be sure that no cases had been ignored. Another example is endgame theory in chess — where machines are now unchallenged, due to their ability in 'brute force' analysis.

iv. It is possible to use foundational theories as frameworks — defining a formal theory inside ZFC is a (theoretically) trivial operation, but it would not be very easy to use. More interesting is the approach suggested in [10], of using the type theory of *Nuprl* as a practical framework.

v. It is not actually necessary that facilities be explicitly available to do this, all that is required is that the proof be constructive; then, if there is no function

definition facility available, instead of normalising a function application, the existence proof itself can be normalised in the same sort of way — though this would be a much less economical operation.

vi. It is worth noting that the idea that meta-theory (or its equivalent) should be constructive is actually 'classical': dating back to Hilbert [42], who proscribed 'infinitistic' methods (though he did not explicitly endorse constructive methods) for philosophical reasons. But it has been proposed more recently for pragmatic reasons similar to what is given here, in, for instance, [8].

vii. For instance, bugs have been found in the decision procedures of the *Nuprl* PDS, which made the logic inconsistent. Because of this, a project to modify the system so that it is possible to replace the derivations of these extra rules with proper derivations has been started. So that when the user is finished the work of interactive proof construction, and the time needed to do this is not so important, it is possible to replace the appeals to the decision procedures with primitive derivations.

viii. In practice, one must add that this is probably not going to be a very useful function, since unless special care is taken a function extracted from a classical proof will be enormously inefficient.

ix. Important exceptions to this are programming language interpreters themselves, which need Turing completeness and (perhaps more relevant here) proof normalisation procedures and the like, which may need induction principles available only in arbitrarily strong theories.

x. For example, when programming in PRA as the system is presented in, say, [47] directly, is simply horrible; unless the structure of the intended computation is of a very particular kind, various sorts of contortions with bounds for functions have to be performed. (The problem of computing with primitive recursive functions is actually discussed further on in this thesis.)

The theory FS_0 as a Logical Framework

The previous chapter looked at what is necessary or desirable in a theory that is to be used as a logical framework, especially one to be used particularly for meta-level reasoning. This chapter describes the particular framework theory that has been used here and how it works. Two alternatives are presented above as possible approaches to building a framework: either type-theoretic, or Post-style. The decision has been made to to use a Post-style theory proposed by Feferman, called FS_0. This chapter describes and explores FS_0, and my implementation and support, then develops a simple example and an initial evaluation.

§1 The theory FS_0

A brief description of FS_0 is that it is a conservative extension of PRA. (However there are at least two definitions of the theory of primitive recursive arithmetic in the literature, so it is important to say exactly which one is meant: here, and in all future discussion, PRA is the theory of arithmetic with same axioms and language (i.e., with the functions and predicates $\succ$ $(\cdot)$, $+$, $\times$ and $<$) as Peano arithmetic but with induction restricted to Σ^0_1-formulae only). Specifically, it is a theory of s-expressions (that is, ordered pairs of objects) and classes of, and primitive recursive functions on, s-expressions which resembles *Pure Lisp*, embedded in a second order predicate logic.

The difference between FS_0 and the theories proposed by Post and Smullyan is twofold: first, Feferman has been able to learn from the practical experience of the last thirty years that we have had, programming real machines instead of thinking about abstract ones, and secondly, he has implicitly supplied a programming facility built into the language. The programming language Lisp, which uses s-expressions as its fundamental data structure, has been in use since the early nineteen sixties, and is still by far the most popular for work in symbolic computation; this is a strong argument in favour of using s-expressions as the foundation of a theory intended for symbolic manipulation.

§2 A formal description of FS_0

This section is a formal description of the language and theory FS_0. The theory is embedded in a sorted predicate logic with equality and membership. There are three sorts, s-expressions, functions and classes, and the following notational conventions have been adopted: $\boldsymbol{S}, \boldsymbol{S_1}, \boldsymbol{S_2}$ vary over s-expressions in the language of FS_0. $\boldsymbol{F}, \boldsymbol{F_1}, \boldsymbol{F_2}$ vary over functions in the language of FS_0. $\boldsymbol{C}, \boldsymbol{C_1}, \boldsymbol{C_2}$ vary over classes in the language of FS_0. $\boldsymbol{Fm}, \boldsymbol{Fm_1}, \boldsymbol{Fm_2}$ vary over formulae in the language of FS_0, and $\boldsymbol{H}, \boldsymbol{H_1}, \boldsymbol{H_2}$ vary over lists of formulae in the language of FS_0. $\boldsymbol{v}, \boldsymbol{v_1}, \boldsymbol{v_2}$ vary over s-expression variables, and $\boldsymbol{c}, \boldsymbol{c_1}, \boldsymbol{c_2}$ vary over class variables.

2.1 The language $\mathcal{L}_{FS_0}$

S-expressions are very similar to the similarly named data structure of *Pure Lisp*; they are defined formally as the smallest set such that:

- O, a constant of sort 'S-expression'.
- variables of type 'S-expression' (such as e.g., $x, y, \ldots, z$ — though this should not be taken as a 'legal' definition).
- If $\boldsymbol{F}$ is a function, and $\boldsymbol{S}$ is of sort 'S-expression', then $\boldsymbol{FS}$ is of sort 'S-expression'. Sometimes a pair of brackets will be added to aid readability, thus $\boldsymbol{F}(\boldsymbol{S})$ is the same as $\boldsymbol{FS}$; this does not introduce ambiguity into expressions such as $\boldsymbol{F}(\boldsymbol{S_1}, \boldsymbol{S_2})$, which is an application of a function to a single s-expression, the pair $(\boldsymbol{S_1}, \boldsymbol{S_2})$.
- If $\boldsymbol{S_1}, \boldsymbol{S_2}$ are of sort 'S-expression', then $(\boldsymbol{S_1}, \boldsymbol{S_2})$ is an s-expression (here the comma can be thought of as a binary function in infix notation).

From now on, for the sake of convenience and readability, the comma is taken as associating to the left, so that $(\boldsymbol{S_1}, \boldsymbol{S_2}, \boldsymbol{S_3})$ is taken as equivalent to $((\boldsymbol{S_1}, \boldsymbol{S_2}), \boldsymbol{S_3})$, rather than $(\boldsymbol{S_1}, (\boldsymbol{S_2}, \boldsymbol{S_3}))$ (beware that this is the other way around from the abbreviation used in *Lisp*).

All functions are unary, and they are defined as follows:

- I, π_1, π_2, D are all functions (to suggest 'Identity', 'Projection Left', 'Projection Right' and 'Decide').
- variables and constants of sort 'Function'.
- If $\boldsymbol{S}$ is an s-expression, then $K_{\boldsymbol{S}}$ is a function (suggesting 'Constant').
- If $\boldsymbol{F_1}, \boldsymbol{F_2}$ are functions, then $\mathcal{C}[\boldsymbol{F_1}, \boldsymbol{F_2}], \mathcal{P}[\boldsymbol{F_1}, \boldsymbol{F_2}], \mathcal{R}[\boldsymbol{F_1}, \boldsymbol{F_2}]$ are functions (to suggest 'Compose', 'Pair' and 'Recursion').

From now on, for the sake of convenience and readability, the following abbreviations will be used:

$$\mathcal{C}[\boldsymbol{F_1}, \ldots, \boldsymbol{F_{n-1}}, \boldsymbol{F_n}] \equiv \mathcal{C}[\mathcal{C}[\boldsymbol{F_1}, \ldots \boldsymbol{F_{n-1}}], \boldsymbol{F_n}]$$

$$\mathcal{P}[\boldsymbol{F_1}, \ldots, \boldsymbol{F_{n-1}}, \boldsymbol{F_n}] \equiv \mathcal{P}[\mathcal{P}[\boldsymbol{F_1}, \ldots \boldsymbol{F_{n-1}}], \boldsymbol{F_n}]$$

$$\pi_{i_1 \cdots i_n} \equiv \mathcal{C}[\pi_{i_1}, \ldots, \pi_n]$$

Classes (which can be thought of, intuitively, as recursively enumerable) are defined as follows:

- $\{O\}$ is a class.
- variables of sort 'Class'.

- If C_1, C_2 are classes, then $C_1 \cup C_2$ and $C_1 \cap C_2$ are classes.
- If F is a function, and C is a class, then $F^{-1}C$ is a class (for inverse image).
- If C_1, C_2 are classes, then $\mathcal{I}_2(C_1, C_2)$ is a class.

When it might make things clearer (especially when using multicharacter identifiers) , the convention of marking class variables, etc., with a subscript $_C$ will be used, similarly functions are marked with a subscript $_F$. Further, use is often made of definitional abbreviations of constant s-expressions, functions and classes in the language, and the same conventions are observed for these. Another convention sometimes used is that constant s-expressions are written as quoted strings of characters.

2.2 The theory FS_0

The theory of FS_0 is given here in a sequent calculus formulation, as an extension of a presentation of a weak second order predicate logic with cut and equality for first order terms (and a definition mechanism for ground terms), rather than the axiomatic presentation used by Feferman, since this is more suitable for implementation as a top down proof development system. By default the hypothesis lists of the sub-goals will be the hypothesis list of the goal. Also, it should be pointed out that rather than the unary negation connective, falsum ($\bot$) is used, i.e., 'not A' is rendered as '$A \rightarrow \bot$' rather than '$\neg A$'.

The first rules define the s-expression pairing (comma) and projection (π_1 and π_2) functions, so that

$$\boldsymbol{H}, (\boldsymbol{S_1}, \boldsymbol{S_2}) = 0 \vdash \bot \qquad \textit{error}$$

and

$$\boldsymbol{H} \vdash \pi_i(\boldsymbol{S_1}, \boldsymbol{S_2}) = \boldsymbol{S_i} \textit{ where } \boldsymbol{i} \in \{1,2\} \qquad p_i$$

which simply state that no pair is equal to O, and the two projection functions do what you imagine they do (notice that the behaviour of the projecton functions applied to O is not defined in the theory).

The rules for the identity (I) and constant (K_{S_1}) functions are exactly what one would imagine:

$$\boldsymbol{H} \vdash \boldsymbol{I}\boldsymbol{S} = \boldsymbol{S} \qquad \textit{Id}$$

and

$$\boldsymbol{H} \vdash K_{S_1} \boldsymbol{S_2} = \boldsymbol{S_1} \qquad \textit{const}$$

The compare (D) function takes an s-expression that can be resolved into a quadruple, and compares the first two terms; if they are equal, then it is equal to the third term, as follows

$$\frac{\boldsymbol{H} \vdash \boldsymbol{S_1} = \boldsymbol{S_2}}{\boldsymbol{H} \vdash D(\boldsymbol{S_1}, \boldsymbol{S_2}, \boldsymbol{S_3}, \boldsymbol{S_4}) = \boldsymbol{S_3}} \quad Dec_t$$

if, on the other hand, they are not equal, then it returns the fourth:

$$\frac{\boldsymbol{H} \vdash \boldsymbol{S_1} \neq \boldsymbol{S_2}}{\boldsymbol{H} \vdash D(\boldsymbol{S_1}, \boldsymbol{S_2}, \boldsymbol{S_3}, \boldsymbol{S_4}) = \boldsymbol{S_4}} \quad Dec_f$$

The third case is when D is applied to an s-expression that cannot be resolved into the appropriate form; i.e., the argument is 'not well formed'. In this case, it simply returns O:

$$\frac{\boldsymbol{H},\ \boldsymbol{S} = (\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{v_4}) \vdash \bot}{\boldsymbol{H} \vdash D\boldsymbol{S} = O} \quad Dec_\bot$$

(where $\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{v_4}$ are new variables).

Then the $\mathcal{C}$ and $\mathcal{P}$ function combinators are ways of combining functions together in new ways; $\mathcal{C}$ composes functions:

$$\boldsymbol{H} \vdash \mathcal{C}[\boldsymbol{F_1}, \boldsymbol{F_2}]\boldsymbol{S} = \boldsymbol{F_1}(\boldsymbol{F_2}\boldsymbol{S}) \quad Cp$$

and $\mathcal{P}$ constructs an s-expression of the result of applying the two functions to the argument:

$$\boldsymbol{H} \vdash \mathcal{P}[\boldsymbol{F_1}, \boldsymbol{F_2}]\boldsymbol{S} = (\boldsymbol{F_1}\boldsymbol{S}, \boldsymbol{F_2}\boldsymbol{S}) \quad Pr$$

The rules for the $\mathcal{R}$ combinator are more complex; it is used to define primitive recursive functions on s-expressions. (In what immediately follows here $\boldsymbol{F_3} \equiv \mathcal{R}[\boldsymbol{F_1}, \boldsymbol{F_2}]$). At first glance the rules seem slightly asymmetrical, but this is only because only one parameter is allowed. The left hand part of the s-expression contains a parameter, while the left hand part contains the s-expression on which the recursion is performed. So the base case of the function is when the right hand side of the argument is simply a O, when the value of $\boldsymbol{F_1}$, the function defining the behaviour in the base case, is applied to the constant parameters on the left. The rule for this circumstance is:

$$\boldsymbol{H} \vdash \boldsymbol{F_3}(\boldsymbol{S}, O) = \boldsymbol{F_1}\boldsymbol{S} \quad Rec_B$$

When, on the other hand, the left hand part can be resolved into an s-expression, then $\boldsymbol{F_2}$, the step case function is applied, as the next rule describes. This is quite

complicated, because, in general, the step case can take account of the constant parameters, and both the two parts of the original s-expression, and the results of the recursive call.

$$\boldsymbol{H} \vdash \boldsymbol{F_3}(\boldsymbol{S_1}, (\boldsymbol{S_2}, \boldsymbol{S_3})) = \boldsymbol{F_2}(\boldsymbol{S_1}, \boldsymbol{S_2}, \boldsymbol{S_3}, \boldsymbol{F_3}(\boldsymbol{S_1}, \boldsymbol{S_2}), \boldsymbol{F_3}(\boldsymbol{S_1}, \boldsymbol{S_3})) \qquad Rec_S$$

The third rule is like the third rule for D, it takes account of the circumstances when a function defined using $\mathcal{R}$ is applied to an argument that is not 'well formed', i.e., O:

$$\boldsymbol{H} \vdash \boldsymbol{F_3} O = O \qquad Rec_\perp$$

The rules for class membership can now be given (the double line here is simply an abbeviation indicating a two-way rule). The first of these is for the only class that is explicitly defined, $\{O\}$, and is obvious.

$$\frac{\boldsymbol{H} \vdash \boldsymbol{S} \in \{O\}}{\boldsymbol{H} \vdash \boldsymbol{S} = O} \qquad Mem_B$$

Obvious too, are the rules defining disjunction and conjunction

$$\frac{\boldsymbol{H} \vdash \boldsymbol{S} \in \boldsymbol{C_1} \cap \boldsymbol{C_2}}{\boldsymbol{H} \vdash \boldsymbol{S} \in \boldsymbol{C_1} \wedge \boldsymbol{S} \in \boldsymbol{C_2}} \qquad Int$$

and

$$\frac{\boldsymbol{H_1} \vdash \boldsymbol{S_1} \in \boldsymbol{C_1} \cup \boldsymbol{C_2}}{\boldsymbol{H_1} \vdash \boldsymbol{S_1} \in \boldsymbol{C_1} \vee \boldsymbol{S_1} \in \boldsymbol{C_2}} \qquad Uni$$

The function inverse constructor is more interesting. It returns the domain of a function, given the class that defines the range.

$$\frac{\boldsymbol{H} \vdash \boldsymbol{S} \in \boldsymbol{F}^{-1}\boldsymbol{C}}{\boldsymbol{H} \vdash \boldsymbol{F}\boldsymbol{S} \in \boldsymbol{C}} \qquad Image$$

Finally, here are the rules for $\mathcal{I}_2(\cdot,\cdot)$. This is used to define recursively enumerable sets under the closure principles below. Given that $\boldsymbol{C_1} \equiv \mathcal{I}_2(\boldsymbol{C_2}, \boldsymbol{C_3})$, the rules are as follows:

$$\boldsymbol{H}, \boldsymbol{S} \in \boldsymbol{C_2} \vdash \boldsymbol{S} \in \boldsymbol{C_1} \qquad Inc_B$$

$$\frac{\boldsymbol{H_1} \vdash \boldsymbol{S_2} \in \boldsymbol{C_1} \qquad \boldsymbol{H_2} \vdash \boldsymbol{S_3} \in \boldsymbol{C_1} \qquad \boldsymbol{H_3} \vdash (\boldsymbol{S_1}, \boldsymbol{S_2}, \boldsymbol{S_3}) \in \boldsymbol{C_3}}{\boldsymbol{H_1}, \boldsymbol{H_2}, \boldsymbol{H_3} \vdash \boldsymbol{S_1} \in \boldsymbol{C_1}} \qquad Inc_S$$

Before giving the induction rules, $C_1 \subset C_2$ is defined as an abbreviation for $\forall v_1 (v_1 \in C_1 \rightarrow v_1 \in C_2)$ where v_1 is some variable that does not occur free in C_1 or C_2. Then induction over a class $C_1 \equiv \mathcal{I}_2(C_2, C_3)$ is:

$$\frac{H_1 \vdash C_2 \subset C_4 \qquad H_2, v_2 \in C_4, v_3 \in C_4, (v_1, v_2, v_3) \in C_3 \vdash v_1 \in C_4}{H_1, H_2 \vdash C_1 \subset C_4} \; ClassInd$$

(where v_1, v_2, v_3 are s-expression variables that do not occur in C_1, C_4). There is also a rule of induction over all s-expressions,

$$\frac{H_1 \vdash O \in C_1 \qquad H_2, v_2 \in C_1, v_3 \in C_1 \vdash (v_2, v_3) \in C_1}{H_1, H_2 \vdash v_1 \in C_1} \; UniInd$$

where v_2, v_3 are new, distinct, variables, that do not occur free in C_1.

§3 Some initial observations

The system described above is very simple, but, like the pure lambda calculus, it is actually quite useable, once a few lemmas have been proven, and a few preliminary definitions have been constructed. This section develops informally some lemmas that are useful in the future, and expands on how some parts of the theory work.

LEMMA 1 (FEFERMAN). *To any formula in FS_0, Fm_1, there corresponds another formula Fm_2 with one free variable over s-expressions, where $\vdash Fm_2$ if and only if $\vdash Fm_1$.*

PROOF: this follows by induction on the number of free variables. If v_1 and v_2 are free variables in Fm_1 then $\vdash Fm_1$ if and only if $\vdash Fm_1[\pi_1 v_1, \pi_2 v_1 / v_1, v_2]$, which has one less free variable over s-expressions.

After this, the first thing to address is the notion of comprehension; for what formulae is it possible to find a class which contains only the objects that satisfy that formula; i.e. for which formulae Fm, with a free variable (say x), is it the case that

$$\exists C \forall x (x \in C \leftrightarrow Fm)$$

and the result for this is in two parts, as follows.

LEMMA 2 (FEFERMAN). *Given an open formula Fm in FS_0 constructed from conjunctions and disjunctions, equalities and inequalities, with free variables $F_1, F_2, \ldots$, then there is a closed class C such that $(F_1, F_2, \ldots) \in C$ iff Fm.*

PROOF: Given a formula $\boldsymbol{Fm}_1$ of the appropriate sort, by lemma 1 there is a equiprovable formula $\boldsymbol{Fm}_2$ with one free variable, $\boldsymbol{v}$. First, replace any s-expression made up solely from O and commas, $\boldsymbol{S}$, with $K_{\boldsymbol{S}}\boldsymbol{v}$, which is equiprovable. Then, while possible, replace some term of the form

$$(\boldsymbol{F}_1\boldsymbol{v}, \boldsymbol{F}_2\boldsymbol{v}) \qquad or \qquad \boldsymbol{F}_1(\boldsymbol{F}_2\boldsymbol{v})$$

with, respectively,

$$\mathcal{P}[\boldsymbol{F}_1, \boldsymbol{F}_2]\boldsymbol{v} \qquad or \qquad \mathcal{C}[\boldsymbol{F}_1, \boldsymbol{F}_2]\boldsymbol{v}$$

which are equiprovable. Then replace all equalities of the form

$$\boldsymbol{F}_1\boldsymbol{v} = \boldsymbol{F}_2\boldsymbol{v} \qquad or \qquad \boldsymbol{F}_1\boldsymbol{v} \neq \boldsymbol{F}_2\boldsymbol{v}$$

with, respectively

$$\mathcal{C}[D, \mathcal{P}[\boldsymbol{F}_1, \boldsymbol{F}_2, K_O, K_{(O,O)}]]\boldsymbol{v} = O \qquad or \qquad \mathcal{C}[D, \mathcal{P}[\boldsymbol{F}_1, \boldsymbol{F}_2, K_{(O,O)}, K_O]]\boldsymbol{v} = O$$

which are equiprovable. Then replace all equalities of the form

$$\boldsymbol{F}\boldsymbol{v} = O \qquad or \qquad O = \boldsymbol{F}\boldsymbol{v}$$

with

$$\boldsymbol{v} \in \boldsymbol{F}^{-1}\{O\}$$

which is equiprovable. Then, finally, repeatedly replace subformulae of the form

$$\boldsymbol{v} \in \boldsymbol{C}_1 \wedge \boldsymbol{v} \in \boldsymbol{C}_2 \qquad or \qquad \boldsymbol{v} \in \boldsymbol{C}_1 \vee \boldsymbol{v} \in \boldsymbol{C}_2$$

with, respectively,

$$\boldsymbol{v} \in \boldsymbol{C}_1 \cap \boldsymbol{C}_2 \qquad or \qquad \boldsymbol{v} \in \boldsymbol{C}_1 \cup \boldsymbol{C}_2$$

which are equiprovable. This will result in a formula of the form $\boldsymbol{v} \in \boldsymbol{C}$ which is provable if and only if $\boldsymbol{Fm}_1$ is provable.

LEMMA 3 (FEFERMAN). *There is comprehension for formulae of the form*

$$\exists x_1, \ldots, x_n \boldsymbol{Fm},$$

where $\boldsymbol{Fm}$ *is open.*

PROOF: The proof for $n = 1$ is given; this generalises obviously. If $x, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ are the free variables of $\boldsymbol{Fm}_1$, then need to prove:

$$\exists C\,((\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n) \in C \leftrightarrow \exists x\,\boldsymbol{Fm}).$$

First define two distinct s-expression constants, call them '0' and '1', then two classes $\boldsymbol{C}_1$, $\boldsymbol{C}_2$ such that:

$$x \in \boldsymbol{C}_1 \leftrightarrow \pi_1 x = \text{'0'}$$

$$(w, z, z) \in \boldsymbol{C}_2 \leftrightarrow y = (\boldsymbol{v}_{11}, \ldots, \boldsymbol{v}_{1n}) \wedge w = (\text{'1'}, y) \wedge z = (\text{'0'}, (x, y)) \wedge \boldsymbol{Fm}$$

Then

$$\exists x\,\boldsymbol{Fm} \leftrightarrow (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n) \in \mathcal{P}[K_{\text{'1'}}, I]^{-1}\mathcal{I}_2(\boldsymbol{C}_1, \boldsymbol{C}_2)$$

3.1 Simple inductive sets

With the lemmas above, it is possible to explain how the $\mathcal{I}_2(\cdot, \cdot)$ class constructor works in general. This allows a class $\boldsymbol{C}_1$ to be defined as the closure of a class $\boldsymbol{C}_2$, under a two place rule

$$\frac{\boldsymbol{v}_2 \in \boldsymbol{C}_1 \qquad \boldsymbol{v}_3 \in \boldsymbol{C}_1}{\boldsymbol{v}_1 \in \boldsymbol{C}_1}\ \boldsymbol{Fm}(\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3)$$

where $\boldsymbol{Fm}$ is a Σ_0^1 relation as above, with free variables $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ and $\boldsymbol{v}_3$, defining the relationship between $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ and $\boldsymbol{v}_3$. If $\boldsymbol{C}_3$ is the extension of $\boldsymbol{Fm}$, we have

$$(\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3) \in \boldsymbol{C}_3 \leftrightarrow \boldsymbol{Fm}$$

Then, it is clear that $\boldsymbol{C}_1 \equiv \mathcal{I}_2(\boldsymbol{C}_2, \boldsymbol{C}_3)$.

In any particular instance of a rule (i.e., where $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3 = \boldsymbol{S}_1, \boldsymbol{S}_2, \boldsymbol{S}_3$), then $\boldsymbol{S}_3$ is said to *depend* on $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ since it is in the class if they are. Notice that FS_0 only allows the definition of classes of objects that depend on at most two earlier objects; this would at first glance seem to be a problem, but Feferman has shown (quoted further on) that this is not the case.

§4 Evaluating expressions

A lot of the work that is carried out in FS_0 in practice is essentially the reduction of function applications. And the work presented here needs regularly to make use of the fact that a useful computation mechanism is available in the system.

However, while Feferman's presentation describes a facility for defining primitive recursive functions, it does not discuss how it 'moves'. As the theory is described, the only way to evaluate a function application automatically is to use a tactic that reduces function applications by algebraic manipulation through the theory. Such an approach is not practical, and is clearly not what Feferman intends. The theory has been augmented therefore, with a computation rule which can evaluate a function application in one step.

Function evaluation is defined by equality on terms, and it is possible to construct a (lazy) evaluation mechanism based on this that reduces terms in the natural way. Furthermore, since it is possible to prove strong normalisation on terms, and that equality between terms is Church-Rosser, the evaluation mechanism will produce the same result as the tactic alternative described above, only a great deal faster. Strong normalisation follows by showing that the maximum total number of reductions possible at any stage is finite. The Church-Rosser diamond property follows by induction on the structure of terms — the proof is much simplified by the fact that all functions are in normal form, so making composed functions much easier to deal with.

$$\boldsymbol{FS_1 \rhd S_2}$$

will sometimes be used to denote $\vdash \boldsymbol{FS_1 = S_2}$.

§5 Some simple examples of FS_0 in use

Now some examples are given of the definitions of simple useful classes. Even before this, though, it is worth defining a couple of constant s-expressions, for true and false. In the lemma above, the constant O was used, for convenience' sake to stand for 'true', while (O,O) stood for 'false' (in fact anything other than O stands for false). So these definitions will be made explicit as

$$true \triangleq O$$

$$false \triangleq (O,O)$$

(where $\triangleq$ is used to indicate definitional abbreviation).

5.1 Equality as a function and as a class

This is a simple example that shows off the use of most of the basic functions and combinators. In the lemma above, a function that tested for equality was constructed from the D function, returning true if the first two parameters are equal, false otherwise. What we need is a simpler function than D that takes a pair and evaluates appropriately on application:

$$equal_F \triangleq \mathcal{C}[D, \mathcal{P}[I, K_{true}, K_{false}]]$$

So that $equal_F(\boldsymbol{S_1}, \boldsymbol{S_2}) \rhd D(\boldsymbol{S_1}, \boldsymbol{S_2}, true, false)$ which is exactly what is needed.

This is the naïve definition of 'equal,' but it is not the most useful here. A lot of the time in FS_0 the user is interested in classes, not functions, and if this is the definition used in practice there will be a lot of class definitions of the form:

$$something_C \triangleq \mathcal{C}[equal_F, \mathcal{P}[\boldsymbol{F_1}, \boldsymbol{F_2}]]^{-1}\{true\}$$

It is usually much better, therefore, to have a definition of the class of equal pairs. This can be done directly from the function using the inverse operation on classes, defining the class as the set of all pairs for which an application of $equal_F$ evaluates to true:

$$equal_C \triangleq {equal_F}^{-1}\{true\}$$

And this definition keeps the useful property of $equal_C$ that it makes use only of the operators that allow construction of (primitive) recursive sets, so membership in $equal_C$ is decidable in the same way as $equal_C$.

5.2 The cross product of two classes (Feferman)

A class often needed in practice is the cross product of two previously defined classes, and this is fairly easily to construct, though not quite as easily as first impression suggests. The intuitive definition of the cross product of classes $\boldsymbol{C_1}$ and $\boldsymbol{C_2}$ is that it is the class of s-expressions $\boldsymbol{S}$ such that $\pi_1 \boldsymbol{S} \in \boldsymbol{C_1}$ and $\pi_2 \boldsymbol{S} \in \boldsymbol{C_2}$. So a first attempt to define it might be:

$$\langle \boldsymbol{C_1} \rangle * \langle \boldsymbol{C_2} \rangle \triangleq {\pi_1}^{-1}\boldsymbol{C_1} \cap {\pi_2}^{-1}\boldsymbol{C_2}.$$

(The angle brackets here are a notation that will be used occasionally in future to indicate that the definition is not of a constant, but instead takes arguments. Such

a definition is not a part of the proper language, only particular instances, with constants substituted for the parameters, are).

Unfortunately this does not take account of the undefined behaviour of π_1 and π_2 when applied to O; it is possible that $\pi_1 O$ and $\pi_2 O$ are equal to values in C_1 and C_2, so that O would be in the cross product defined this way. This is not so improbable as perhaps it might seem; imagine that the class $\{(O,O)\}$ is wanted. This could be defined as the inverse of the function $f_F = C(D,[K_O,\pi_1,\pi_2],K_{false}])$ which looks to see if the π_1 and the π_2 of an s-expression α are equal to O. But, as was explained above, projections of O are undefined in the theory. An implementation $\mathfrak{A}$ of the system might have $\models \pi_1 O = O$ and $\models \pi_2 O = O$; if this were the case then $\models f_F O = D(\pi_1 O, O, \pi_2 O, false) = D(O,O,O,false) = O$, so $\models O \in f_F^{-1}\{O\}$. Which, while perfectly reasonable, is not what was wanted. The problem can be avoided as follows:

$$pairs_C \triangleq \mathcal{P}[\pi_1,\pi_2,I]^{-1}\, equal_C$$

$$\langle C_1\rangle \times \langle C_2\rangle \triangleq \langle C_1\rangle * \langle C_2\rangle \cap pairs_C$$

since

$$(\pi_1 O, \pi_2 O) \neq O$$

5.3 The class of lists of objects of a class C.

The examples above are simple demonstrations of how to use FS_0, but they are really only preliminary to the sort of definitions that the theory is intended for. A more substantial example, that is useable in itself, is the class of lists: here the class of lists of members of the abstract class C is defined.

The empty list can be represented as $\langle\rangle = O$ (in the same manner as pure *Lisp*), and a list of objects of a class C as an s-expression S where $\pi_1 S$ is itself a list of members of C, and $\pi_2 S$ is in C (notice that the list is reversed compared to the equivalent in *Lisp*); a recursive definition dependent on one previous object. But, since in FS_0 definitions of recursively enumerable classes are given using relation triples, even though only relation pairs are needed, the easiest approach is to define triples, and ignore one argument. The triple can be defined as the class C, which is the solution to

$$(v_1,v_2,v_3,v_4) \in C \leftrightarrow v_3 = v_1 \wedge v_2 \in C$$

(notice that, while $\boldsymbol{v_4}$ is mentioned on the left, it does not occur on the right so, in effect, no constraints are placed on what goes into this 'slot' of any acceptable s-expression). Then one possible definition is $List'\langle\cdot\rangle$ below, which in turn allows lists to be defined schematically.

$$List'\langle \boldsymbol{C}\rangle \triangleq \mathcal{P}[\pi_{11},\pi_2]^{-1}\, equal_C \cap {\pi_{21}}^{-1}\boldsymbol{C} \cap {\pi_1}^{-1} pairs_C$$
$$List\langle \boldsymbol{C}\rangle \triangleq \mathcal{I}_2(\{O\}, {\pi_1}^{-1} List'\langle \boldsymbol{C}\rangle)$$

Notice how the second parameter of the relation in $List\langle \boldsymbol{C}\rangle$ is ignored, and the apparently redundant extra conjunct, $pairs_C$, in the definition of $List'\langle\cdot,\cdot\rangle$ — the reason for this is discussed and explained below.

Given this definition, it is a straightforward matter to prove an induction theorem:

$$\begin{aligned}\forall C\,\forall D\,(O \in D \rightarrow \forall x\,\forall y\,(x \in D \rightarrow y \in C \rightarrow (x,y) \in D)\\ \rightarrow List\langle C\rangle \subset D)\end{aligned} \qquad (LI)$$

First, removing leading quantifiers and implications reduces the problem to

$$O \in D,\ \forall x\,\forall y\,(x \in D \rightarrow y \in C \rightarrow (x,y) \in D) \vdash List\langle C\rangle \subset D$$

at which point it is possible to apply the rule for induction over classes, *ClassInd*, which produces two subgoals. The trivial base case

$$O \in D \vdash \{O\} \subset D$$

and the step case

$$\begin{aligned}&\forall x\,\forall y\,(x \in D \rightarrow y \in C \rightarrow (x,y) \in D),\\ &z \in D,\\ &w \in D,\\ &(v,w,z) \in {\pi_1}^{-1} List'\langle D\rangle\\ &\quad \vdash v \in D\end{aligned}$$

Instantiating the hypothesis and simplifying expressions reduces this to

$$\begin{aligned}&w \in D \rightarrow \pi_2 v \in C \rightarrow (w, \pi_2 v) \in D,\\ &w \in D,\\ &(v,w) \in List'\langle D\rangle\\ &\quad \vdash v \in D.\end{aligned}$$

Then, by cutting in as follows,

$$\begin{aligned} & w \in D \to \pi_2 v \in C \to (w, \pi_2 v) \in D, \\ & w \in D, \\ & \pi_1 v = w \wedge \pi_2 v \in C \wedge v = (\pi_1 v, \pi_2 v) \\ & \qquad \vdash v \in D. \end{aligned}$$

the rest of the proof is trivial. Notice that it is only possible to show that $v = (\pi_1 v, \pi_2 v)$ because it is explicitly stated in the definition of $List'\langle\cdot\rangle$ that this is so. The need for extra constraints beyond what are obviously needed for a definition is a recurring feature of FS_0 about which care has to be taken (I will discuss this again when I consider the sorts of tools that I have defined for the theory). This leaves only the cut

$$(v, w) \in List'\langle D\rangle \vdash \pi_1 v = w \wedge \pi_2 v \in C \wedge v = (\pi_1 v, \pi_2 v)$$

to be justified, and this follows almost directly from the definition of $List'\langle\cdot\rangle$.

5.4 The natural numbers

Using the schematic list definition above as a skeleton, it is easy to define a class that encodes the natural numbers, $\mathbb{N}$. This can be defined as the class of lists of members of the class $\{O\}$; i.e.,

$$nat_C \triangleq List\langle\{O\}\rangle$$

so that

$$\begin{aligned} \overline{0} &= O \\ \overline{n+1} &= (\overline{n}, O) \end{aligned}$$

then (primitive) induction over the naturals is an instantiation, and simplification, of LI above:

$$\forall C\,(O \in C \to \forall \boldsymbol{v_1}\,(\boldsymbol{v_1} \in C \to (\boldsymbol{v_1}, O) \in C) \to nat_C \subset C) \qquad (NI)$$

5.5 List membership

So far, some trivial functions have been defined. But there has been no function definition so far that has used recursion. A simple recursive relation that is naturally defined as a function, rather than as a class of pairs, is list membership.

What is needed is a function that, when applied to a pair consisting of an object and a list of objects (for convenience, on the left and right respectively), evaluates to true iff the object occurs on the list. A set of defining equations for this is

$$member_F(\boldsymbol{S}, O) = false$$
$$member_F(\boldsymbol{S}_1, (\boldsymbol{S}_2, \boldsymbol{S}_3)) = \begin{cases} true & \text{if } \boldsymbol{S}_3 = \boldsymbol{S}_1 \\ member_F(\boldsymbol{S}_1, \boldsymbol{S}_2) & \text{if } \boldsymbol{S}_3 \neq \boldsymbol{S}_1 \end{cases}$$

Using the $\mathcal{R}$ combinator, the function for the base case is simply the constant function K_{false}. The step case is more messy; recursion is needed only on the tail of the list (π_1), so the value for the recursion on the right can be thrown away immediately, and then it is merely a case of seeing if the α is equal to the π_1 of the list and returning true, or if that is not the case, then the value of the recursion on the rest of the list, which gives a definition as follows:

$$member_F{}^s \triangleq \mathcal{C}[D, \mathcal{P}[\pi_{1111}, \pi_{211}, K_{true}, \pi_{21}]]$$
$$member_F \triangleq \mathcal{R}[K_{false}, member_F{}^s]$$

which can easily be shown to satisfy the definitions as follows. For the base case

$$\begin{aligned} &member_F(\boldsymbol{S}, O) \\ &\quad = \mathcal{R}[K_{false}, member_F{}^s](\boldsymbol{S}, O) \\ &\quad = K_{false}(\boldsymbol{S}, O) \\ &\quad = false \end{aligned}$$

while for the step case

$$\begin{aligned} &member_F(\boldsymbol{S}_1, (\boldsymbol{S}_2, \boldsymbol{S}_3)) \\ &\quad = \mathcal{R}[K_{false}, member_F{}^s](\boldsymbol{S}_1, (\boldsymbol{S}_2, \boldsymbol{S}_3)) \\ &\quad = member_F{}^s(\boldsymbol{S}_1, \boldsymbol{S}_2, \boldsymbol{S}_3, member_F(\boldsymbol{S}_1, \boldsymbol{S}_2), member_F(\boldsymbol{S}_1, \boldsymbol{S}_3)) \\ &\quad = D(\boldsymbol{S}_1, \boldsymbol{S}_3, K_{true}, member_F(\boldsymbol{S}_1, \boldsymbol{S}_2)) \\ &\quad = \begin{cases} true & \text{if } \boldsymbol{S}_3 = \boldsymbol{S}_1 \\ member_F(\boldsymbol{S}_1, \boldsymbol{S}_2) & \text{if } \boldsymbol{S}_3 \neq \boldsymbol{S}_1 \end{cases} \end{aligned}$$

§6 Further observations etc.

6.1 Mutual recursion

The example above shows how it is possible to define simple primitive recursive functions using the $\mathcal{R}$ combinator; it does not show how to define more complex primitive recursive functions. Imagine a pair of functions defined in terms of each other, i.e.:

$$\begin{aligned}
\boldsymbol{f}_1(x, O) &= f'x \\
\boldsymbol{f}_1(x, (y, z)) &= f''(x, y, z, \\
&\qquad (\boldsymbol{f}_1(x, y), \boldsymbol{f}_2(x, y)), \\
&\qquad (\boldsymbol{f}_1(x, z), \boldsymbol{f}_2(x, z))) \\
\boldsymbol{f}_2(x, O) &= g'x \\
\boldsymbol{f}_2(x, (y, z)) &= g''(x, y, z, \\
&\qquad (\boldsymbol{f}_1(x, y), \boldsymbol{f}_2(x, y)), \\
&\qquad (\boldsymbol{f}_1(x, z), \boldsymbol{f}_2(x, z)))
\end{aligned}$$

(given f' and f'' are already defined). The problem is to construct these functions in FS_0. The solution is analogous to the solution for defining mutually dependent general recursive functions using the fixed point operator [56]:

$$\begin{aligned}
fg &\triangleq \mathcal{R}(\mathcal{P}[f', g'], \mathcal{P}[f'', g'']) \\
f &\triangleq \mathcal{C}[\pi_1, fg] \\
g &\triangleq \mathcal{C}[\pi_2, fg]
\end{aligned}$$

It can be easily seen by induction on s-expressions that this satisfies the definition above. In the base case

$$fg(\boldsymbol{S}_1, O) = (f'\boldsymbol{S}_1, g'\boldsymbol{S}_1)$$

so the definitions of f and g are correct here. In the step case,

$$fg(\boldsymbol{S}_1, (\boldsymbol{S}_2, \boldsymbol{S}_3))$$

if we assume that the recursions on the two subparts of the tree return pairs

$$(f(\boldsymbol{S}_1, \boldsymbol{S}_2), g(\boldsymbol{S}_1, \boldsymbol{S}_2))$$

and

$$(f(\boldsymbol{S}_1, \boldsymbol{S}_3), g(\boldsymbol{S}_1, \boldsymbol{S}_3))$$

then it will be clear, on inspection, that the result is exactly what is needed.

It should be fairly clear how to generalise this method to more complicated definitions. However, this example still deals only with functions defined by primitive recursion. Later on, I describe more general support that I have developed for constructing functions defined using course of values induction. Of course, whatever facilities are supplied, it will still only be possible to define primitive recursive functions.

6.2 Recursively enumerable sets

The purpose of FS_0 is to enable the easy and convenient handling of general r.e. sets of s-expressions, so in a complete description of how to use the system there should be an example of this. One of the first things that is striking about FS_0 is that it only explicitly allows recursive definitions with two dependents, which is, at first sight, very restricting; it turns out not to be a problem though.

Consider an r.e. set $\boldsymbol{C}$ which is defined as the closure of the the base case $\boldsymbol{C}_1$ under the rule

$$\frac{\boldsymbol{v}_1 \in \boldsymbol{C} \quad \cdots \quad \boldsymbol{v}_n \in \boldsymbol{C}}{\boldsymbol{v} \in \boldsymbol{C}}\, \boldsymbol{Fm}$$

(where $\boldsymbol{Fm}_1$ is a Σ_1^0-formula with free variables $\boldsymbol{v}, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$). But it is not possible to define this class directly in FS_0, since only classes with members dependent on at most two previous members are allowed. The solution is as follows; given that $\boldsymbol{C}_2$ is the class such that

$$(O, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_n, \boldsymbol{v}) \in \boldsymbol{C}_2 \leftrightarrow \boldsymbol{Fm}$$

(by the comprehension lemma) and two new distinct constants '*memberlist*' and '*member*' are defined, then it is possible to define the class $\boldsymbol{C}^*$ as the closure of the class $\boldsymbol{C}_1{}^* \equiv \{(\boldsymbol{v}_1, \mathit{member}) \mid \boldsymbol{v}_1 \in \boldsymbol{C}_1\} \cup \{(O, \text{'}\mathit{memberlist}\text{'})\}$, under the rules

$$\frac{(\boldsymbol{v}_1, \text{'}\mathit{memberlist}\text{'}) \in \boldsymbol{C}^*}{(\boldsymbol{v}_2, \text{'}\mathit{member}\text{'}) \in \boldsymbol{C}^*}\, (\boldsymbol{v}_1, \boldsymbol{v}_2) \in \boldsymbol{C}_2$$

and

$$\frac{(\boldsymbol{v}_1, \text{'}\mathit{memberlist}\text{'}) \in \boldsymbol{C}^* \qquad (\boldsymbol{v}_2, \text{'}\mathit{member}\text{'}) \in \boldsymbol{C}^*}{(\boldsymbol{v}_1, \boldsymbol{v}_2, \text{'}\mathit{memberlist}\text{'}) \in \boldsymbol{C}^*}$$

which, since they depend on only one or two previous members, are definable in FS_0 directly. Then it is possible to define C in FS_0 as

$$\boldsymbol{v}_1 \in \boldsymbol{C} \leftrightarrow (\boldsymbol{v}_1, \text{`}member\text{'}) \in \boldsymbol{C}^*$$

This method of labeling the classes is very general in its possible applications, so that arbitrary inductive definitions, or even sets of mutually dependent inductive definitions, can be easily defined.

6.3 Induction on other order relations

In practice, the primitive stepwise induction facility of FS_0 is not as useful as course of values induction on the ordering 'is a subtree of', i.e., the least relation closed under

$$\boldsymbol{S}_1 \prec (\boldsymbol{S}_1, \boldsymbol{S}_2)$$
$$\boldsymbol{S}_2 \prec (\boldsymbol{S}_1, \boldsymbol{S}_2)$$
$$\boldsymbol{S}_1 \prec \boldsymbol{S}_2 \rightarrow \boldsymbol{S}_2 \prec \boldsymbol{S}_3 \rightarrow \boldsymbol{S}_1 \prec \boldsymbol{S}_3.$$

For which it is possible to show that

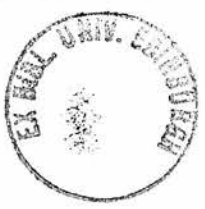
$$\forall C \forall x (\forall y (y \prec x \rightarrow y \in C) \rightarrow x \in C) \rightarrow \forall x (x \in C).$$

The proof of this is really too long to present here, but it is given in the appendices.

6.4 Decidable subtheories

While function applications have a normal form, so that equality between ground s-expressions is decidable, the membership relation between a ground s-expression and a ground class is not (membership in recursively enumerable classes is not decidable in general). However, if a certain subset of FS_0 is used instead, then class definitions become decidable. In fact, membership is decidable for any FS_0 class definition that does not use a $\mathcal{I}_2(\cdot,\cdot)$ construct. The simple proof is by induction on the structure of the definition. Given that $\boldsymbol{C}$ is a ground class definition of the appropriate sort, and $\boldsymbol{S}$ is a ground s-expression, then there is only one base case: $\boldsymbol{S} \in \{O\}$ if and only if $\boldsymbol{S} \rhd O$. For the step case, there are three possibilities:

$$\boldsymbol{C} \equiv \boldsymbol{C}_1 \cup \boldsymbol{C}_2$$
$$\boldsymbol{C} \equiv \boldsymbol{C}_1 \cap \boldsymbol{C}_2$$
$$\boldsymbol{C} \equiv \boldsymbol{F}^{-1}\boldsymbol{C}_1$$



In the first two cases, given that $\boldsymbol{S} \in \boldsymbol{C}, \boldsymbol{S} \in \boldsymbol{C}_1$ are decidable by hypothesis, the conjunction or disjunction is decidable. In the last case, $\boldsymbol{FS} \in \boldsymbol{C}_1$ is decidable by hypothesis, and is true if and only if $\boldsymbol{S} \in \boldsymbol{F}^{-1}\boldsymbol{C}_1$.

Unfortunately, this result is not really as useful as it at first appears, since often it is much more convenient to define a class using $\mathcal{I}_2(\cdot,\cdot)$, even if it could be defined in the decidable subtheory of FS_0.

6.5 Class complement operations

One operation that is noticeable by its absence in the description of FS_0 above is a complement operator; i.e., a constructor that, given a class $\boldsymbol{C}_1$ returns a class $\boldsymbol{C}_2$, where

$$\vdash \boldsymbol{v}_1 \in \boldsymbol{C}_2 \leftrightarrow \boldsymbol{v}_1 \notin \boldsymbol{C}_1$$

If such an operation was available, it would be possible to define many more classes (at least Σ_n-relations, for arbitrary n, or any relation definable in Peano arithmetic). But the interesting classes here are the recursive and the recursively enumerable ones (i.e. Σ_1), which are already available[i].

§7 FS_0 as a framework

The above discussion has been about mathematical properties of FS_0 and basic facilities that it offers for list processing and inductive closure. The question of how it can be used as a framework theory however, has not been addressed. The next section looks at how FS_0 behaves in practice, when used in a system supplying proper support.

It seems that FS_0 is good at handling lists and recursively enumerable sets, and for demonstrating facts about them. It would seem to be able to handle syntax well, and have adequate facilities for proving meta-level assertions about an encoded theory. The place where it seems to be weakest is in what it offers to deal with the peculiarities of logics over other inductively defined sets. That is, it does not directly treat bound variables; rather it is necessary for the user to define what is needed to do that himself. The question is, 'how well the first two properties work to balance the disadvantage of the third?'; but this can only be answered in the light of experience. The rest of this thesis, among other things, describes such experience, which allows some conclusions to be drawn.

§8 An implementation of FS_0

In this section I describe the extensions to the theory and the supporting facilities available in my implementation of FS_0. Then, to complement this, in the next section I discuss the concrete details of the implementation.

8.1 Extensions to the basic logic

The initial set of rules of the system are the standard set for predicate logic with cut, along with the rules listed above which define FS_0. This, though, is not really enough to do real work, so I have extended the basic rule set in various ways simply for convenience sake.

The first of these extensions is to the rules that define FS_0. So that reasoning about assertions in the context of a goal is easier, I have added various 'left' rules. For instance, there is a rule

$$\frac{\boldsymbol{H_1}, \boldsymbol{S} \in \boldsymbol{C_1} \vdash \boldsymbol{Fm_1} \qquad \boldsymbol{H_2}, \boldsymbol{S} \in \boldsymbol{C_2} \vdash \boldsymbol{Fm_1}}{\boldsymbol{H_1}, \boldsymbol{H_2}, \boldsymbol{S} \in \boldsymbol{C_1} \cup \boldsymbol{C_2} \vdash \boldsymbol{Fm_1}}$$

which allows analysis by cases for a class disjunction to be carried out directly, in a way similar to how disjunction in the context is treated by the rules for predicate logic.

The second set of extensions are the substitution rules for formulae and terms. Unlike those mentioned above, which are convenient, but could be replaced with tactics, the substitution rules really are necessary in practice: since they are derived there are tactics corresponding to them, but these are impractically inefficient. The rule for s-expression substitution resembles

$$\frac{\boldsymbol{H_1} \vdash \boldsymbol{S_1} = \boldsymbol{S_2} \qquad \boldsymbol{H_2} \vdash \boldsymbol{Fm_1}[\boldsymbol{S_2}/\boldsymbol{S_1}]}{\boldsymbol{H_1}, \boldsymbol{H_2} \vdash \boldsymbol{Fm_1}}$$

The word 'resembles' is used here, since the implemented rule is actually more complex than this. It allows a user to mark distinguished instances of the term $\boldsymbol{S_1}$ to be replaced, ignoring the others. For instance, given a goal

$$\vdash_{FS_0} (a, a) \in \Gamma$$

it would be possible to invoke a pseudo-rule

$$\mathit{substitute}(([\mathit{sub}(\boldsymbol{S_1})], a) \in \Gamma)$$

which would result in two subgoals being introduced:

$$\vdash_{FS_0} (\boldsymbol{S}_1, a) \in \Gamma$$

and

$$\vdash_{FS_0} \boldsymbol{S}_1 = a$$

Another version of the same rule allows substitution of formulae, requiring logical equivalence instead.

Following this, the other set of important derived rules is for reducing s-expressions to some sort of normal form (these are collectively known as 'compute' rules). Like the substitution rules described above, these allow particular parts of a formula to be marked so that only that part is normalised. Various options are possible, such as reducing a function application only until it has a particular form (e.g., until it is in a state where π_1 or π_2 can be applied to it), or until it has been completely normalised, (i.e., with no function applications at all in it). These rules are complex, since to make them efficient, they have, for example, built in heuristics; and also various facilities that try to make them more useful for when they are applied to non-ground terms. The evaluation mechanism is also 'lazy', i.e., it always reduces the outermost possible redex first, while trying to arrange things so that no extra work is done (there is a risk with a naive outermost redex reduction algorithm, that if, say, a term is substituted into another in several places, then each of the identical copies will have to be reduced in identical manner).

The last of the rule extensions is not really a rule, but better thought of as a syntactic convenience, or an extension of the compute rule. In order to make the system useable a definition mechanism that allows s-expression, function, and class constants to be abbreviated to names proposed by the user has been added (In fact this device has already been used implicitly above, when definitional equality was distinguished from logical equality by the use of the symbol '$\triangleq$' instead of '='). A definition facility like this also needs a facility that can fold ground terms in a formula into constant names, or expand constant names into ground terms *in situ*, and this is what the so-called 'fold' and 'unfold' rules do.

8.2 Simple support facilities

The description above describes the rules of my PDS, but does not discuss the interface. The first important thing to realise here is that *Oyster* [13] is designed as an extension of a *Prolog* interpreter, rather than just programmed in *Prolog*, so the instructions for building derivations are given through the *Prolog* command line. This means that *Prolog* itself is immediately available as a language for automating derivations, i.e., as a tactic language. So it is possible, simply using *Prolog*, to connect together simple rule applications in arbitrary ways.

Unfortunately, simply automatically stringing command line rule applications together means that the only structure in the proof is at the level of primitive rule applications, since these are the only steps that are actually used to build proofs. This means first that the proof is extremely large, and second, that it has lost the structure that the user had in mind in constructing it.

This problem is solved by a combinator for tactics, called a *tactical* [33]. There are several of these, which can be used to build a tactic from others, so that it can be thought of as a sort of compound rule (that is, the system can keep track of its application in the same way as a single rule application). This makes it possible both to keep the size of the derivation down, and to record the 'high level' structure of the proof. The tacticals that *Oyster* supplies are such things as 'repeat', 'or', 'then', etc. which have the obvious meaning. This is more important here, since *Prolog* is relentlessly first order and so building general combinators for procedures is messy and fairly difficult (it involves programming with so-called meta-level predicates, which have few nice properties[ii]). To compensate for this though, the pattern matching facilities of the language are perfectly designed for the sort of symbolic manipulation needed.

8.3 More sophisticated support

The description above leaves one problem with FS_0 unanswered: the directly supplied mechanisms for defining classes and functions in the theory are painfully primitive, in the same sort of way as the mechanisms supplied directly by a computer for use in building software (i.e., the machine code) are painfully primitive. But as with machine code, it is possible to write a compiler and other support software, so that it is not necessary to work directly with the primitive facilities. This is what has been

done, by writing a suite of compilers and cross referencing programs that allow the user to define classes and functions at a higher level without having to worry so much (unlike a compiler for a high level language though, it is not possible to escape the primitive mechanisms completely since in order to work with a definition, its internal structure is needed — though given various tools this need not be a great problem either).

The relation compiler

The first of these support tools is the relation and tuple compiler, which is essentially an implementation of the comprehension lemma presented above. This allows a class that is to be used as a relation in an inductive definition, or as a class itself, to be described in a fairly natural way.

A class is defined by writing a 'stencil' for the sort of s-expressions that it should contain, along with a declaration about what classes the slots in the stencil come from, and what their relation is to one another. Various abbreviations are allowed, such as declaring constants and functions in place, as well as slots that correspond to conjunctions and disjunctions of classes. For instance, a declaration

$$\begin{aligned} foo_C \triangleq\ & \mathsf{tup}\ ((\boldsymbol{v}_1, \boldsymbol{v}_2), f, (\boldsymbol{v}_1, \boldsymbol{v}_2)) \\ & \mathsf{where}\ \boldsymbol{v}_1 \in \boldsymbol{C}_1 \\ & \qquad \boldsymbol{v}_2 \in \boldsymbol{C}_2 \end{aligned}$$

declares a class foo_C which comprehends all s-expressions where the left hand part is a pair of objects from the classes $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$, and the right hand part is the result of applying f to the left. This would result in a definition which, in primitive form, would have been typed in:

$$\begin{aligned} foo_C \triangleq\ & {\pi_{11}}^{-1}\boldsymbol{C}_1 \cap {\pi_{21}}^{-1}\boldsymbol{C}_2 \\ & \cap \mathcal{P}[\pi_2, \mathcal{C}[f, \mathcal{P}[\pi_{11}, \pi_{21}]]]^{-1} equal_C \\ & \cap {\pi_1}^{-1} pairs_C \\ & \cap pairs_C \end{aligned}$$

Notice how the compiler inserts all the extra declarations (the intersection of various parts of the class with the class of pairs) needed to make sure that the tuple really does have the proper structure.

The relations used to define inductive classes are specified in a similar way, but to emphasise that they are to be used to represent rules depending on two previous cases. A typical definition would be written in the form equivalent (though intended for input to a computer) to

$$C_2 \triangleq \frac{v_1 \qquad v_3}{(v_1, v_2)} \quad \text{where } (v_2, v_3) \in C_1$$

If the compiler merely took the description of the class and turned it into a proper FS_0 specification it would not really be a great deal of use, since in order to reason about the structure of the class, the user has to know how it is built. In fact, inconvenient and difficult though class descriptions are to build by hand, they could be argued to have the advantage that the user, having built them, knows how they are put together, and so can easily take them apart. On the other hand, he will have no idea about the structure that has been generated by the compiler (the situation may even be made worse by the several 'optimising' features that have been installed to try to reduce the size of the resulting term). So that this is not a problem, the compiler is also able to remember details of the class definition itself, and how it relates to the structure of the constructed term, and it is possible for tactics to use this information automatically.

The function compiler

The other compiler is for primitive recursive functions. This is needed because the primitive facilities of FS_0 only allow a function f_1 to be built directly if it can be defined using recursion equations that define a value in terms of direct subcomponents; i.e., an application $f_1 v_1$ is has to be defined in terms of $f_1 O$, $f_1(\pi_1 v_1)$ and $f_1(\pi_2 v_1)$. The functions that are needed in practice though are rarely, if ever, sensibly definable this way.

It is a well known fact due to Ackermann [28] that it is not possible to define arbitrary recursive functions using just a primitive recursive combinator. However a far greater range of functions are definable than first impressions suggest; for example, simple course of values recursion, using the ordering relation $\prec$ (or subterm) which was discussed earlier. This is a much more useful facility for defining recursive functions than the simple structural recursion operator allows. If it is used, the only

requirement placed on the function definition is that the result of a function applied to an object must be computable in terms of the function applied to smaller terms, when smaller is defined in terms of '$\prec$'. Also, because there is already a course of values induction theorem proven for exactly this ordering, reasoning about functions based on such an order is easy.

The compiler allows precisely these functions to be defined: it takes as input a set of recursion equations which are well founded according to the subterm relationship, and constructs an function automatically out of the available structural recursion facilities.

The function applied to $\boldsymbol{S}$ is defined in terms of the usual language for function constructions, only extended with a special term $recurse(l)$ which stands for the value of applying the function to that subterm.

For example, a function (or set of functions) defined by the equations:

$$f\ (x, 0) = b\ x$$
$$f\ (x, (a, \text{`}foo\text{'})) = f_{\text{`}foo\text{'}}\ (x, f\ (x, a))$$
$$f\ (x, ((a, b), \text{`}bar\text{'})) = f_{\text{`}bar\text{'}}\ (x, (f\ (x, a), f\ (x, b)))$$

would be given to the compiler in essentially that form. Termination is guaranteed by the restriction on what the function can be applied to: for a function $f(x, a)$ the only applications of f allowed on the right hand side are of the form $f(x, a')$ where $a' \prec a$. This is certainly an improvement on the primitive facility, but it is still restricting — for instance not all recursion equations that have a primitive recursive solution are naturally expressed in this way (as will be seen later).

One possible criticism of the compiler, which is really common to all functions implemented using a primitive recursive combinator, is that it can generate very inefficient code. But given the lazy evaluation mechanism that has been implemented in the compute rules, this is not a problem in practice.

§9 The implementation of FS_0

This section describes the details of the implementation and how well it works, and its effectiveness.

9.1 Oyster and Nuprl

The implementation is a modified version of a proof development system called Oyster designed and built by Christian Horn. This was initially intended to resemble *Nuprl*, and we will briefly describe *Nuprl* and the unmodified form of Oyster first, before discussing the modified system used here.

Like *Nuprl*, Oyster is a PDS for a sequent calculus presentation of Martin-Löf type theory, and while it is intended to have the same 'feel', it is very different in a lot of ways. The distinguishing features of *Nuprl* are that it is a system for developing proofs in a sequent calculus presentation of Martin-Löf type theory and that the tactic language is *ML*(the implementation language is *Lisp*, but this is hidden from the user). Because of the safe and flexible typing of *ML*, the implementors have been able to present the system to the user in very pure terms, as a collection of abstract data-types. There are also good facilities for building and structuring large collections of theorems. All this makes for a very secure, professional system; however it also makes for a very large system — for instance it has to include a complete *ML* interpreter written in *Lisp* — that is measured in megabytes of code, and can run only on fast workstation computers with large memories.

Oyster, on the other hand, is a small system designed to run on personal computers that are not able even to run modern dialects of *Lisp*, never mind *ML*. Its characteristic features, after the logic that it works with, are that it is implemented simply as an extension to a *Prolog* interpreter and that it is a great deal smaller (the source code runs to 7139 lines, or 280 kilobytes). In exchange for this reduction in size, the system is very different; for instance it does not have the same tools for structuring large developments. Much more important than this though is that the data abstraction enforced in *Nuprl* is no longer possible: since the system is an extension of the *Prolog* system, the implementation of the various data types cannot be effectively hidden from the user, and there is thus no way that the same security can be enforced. On the other hand, as a result of this the underlying *Prolog* system can be used directly as the tactic language, and while there may be dispute about the relative merits of *Prolog* and *ML*, there is no denying that *Prolog* is, absolutely, an effective tactic language for a proof development system. A corollary of the smallness of the system is that it is much more easily understandable and modifiable, and this is the reason that it was chosen. Since the system was only intended as an experiment

to be used by one person, issues like security were not seen as important (while it is not difficult to subvert the system, this is unlikely to happen accidentally, especially since the the proofs that it is used to build here are not so subtle that they would hide such a problem if it did arise).

9.2 The patched form of Oyster

The modifications to Oyster can be divided up into three parts: there are the patches to the code of Oyster, there are what might be thought of as 'fundamental' tactics (i.e., code that is essentially part of the basic system, but working through the tactic interface) and, finally, there are ordinary tactics.

Patching the system

While it might seem that modifying the system is a matter of replacing the rules for Martin-Löf type theory with rules for FS_0, in practice it is not that simple. The basic patches amount, in total, to over 3000 lines of code. Beyond the rules, subsystems for rewriting and the lemma mechanism and the the like had to be built or heavily modified. There are also problems with the fact that the system was originally designed to support a very different theory (Martin-Löf type theory), and the special facilities for this, particularly the machinery for keeping track of the witness term of a proof, are not relevant to FS_0, and make every rule application much more complicated than it really needs to be, and certainly slows down the implementation a lot and takes up enormous amounts of room (see below). In a more general way, while the machinery of Oyster did not slow things down, it did make impossible, or very difficult to implement, various ideas for improving the speed and space efficiency.

Fundamental tactics

Apart from the patches to the system itself there are also extensions that, while they are technically tactics, are really part of the basic system in that it is practically unusable without them. These are the various compilers for functions, tuples and classes described above. They consist of another approximately 1000 lines of (very dense) code.

Tactics

Finally, another 5000 lines of tactics was written to support the various developments, most of which were reused with each of the various developments in turn.

9.3 Supporting software and hardware

The software used was versions 2 and 3 of the Quintus *Prolog* interpreter/compiler system running on a various Sun or Sun compatible systems — starting with on a Sun-3 with approximately 20 megabytes of memory, and moving first to a Sun-4 with about 30 megabytes, and finally to a large Solbourne system with 512 Megabytes (although in practice only about 100 Megabytes of this last machine was ever used, even *in extremis*). The runtime size of proof developments usually ranged between 5 and 60 megabytes, though we believe — it is not possible really to say — that part of this size may have been due to leakages in the memory management of Oyster, and some of it was certainly due to unsuitability of Oyster for the application. Certainly, like a lot of software of this sort, it is very vulnerable to paging, and large physical address spaces really are needed.

The question of the run-time performance of the system is tricky to answer since this varied — the performance towards the end of the work was much better than the performance at the beginning (ironically considering the much more powerful machines available later). This was not because the performance of the implementation improved — this was an area where, as was said above, it was difficult to do anything much — but because a lot was learned about how to use the system more efficiently. However the implementation could never be described as fast.

The worst performance was for the development of the prenex normal form theorem: it was usual to wait a couple of minutes after even a simple instruction had been entered before the prompt returned (in fact this theorem was first developed in three or four pieces, since it it would not fit into the physical memory of the machine used to develop it).

The best performance was for the last development that was done — the deduction theorem for propositional logic, where the work was much better structured and as a consequence, for instance, the user never had to wait for single steps of the derivation. Also, tactics were designed so that when a lot of work was needed, it tended to happen in very large chunks, so that the user had time to do something

else instead of passively waiting on the machine.

9.4 The various developments

There are four developments in the text, but they really divide into three parts, since the prenex normal form theorem and the reflection discussion are closely related. I will outline each one separately.

The deduction theorem

This was the last, even if it was the simplest, development, and made use of all the experience of doing the others, and the various supporting software that had been built. The specification of the theory was very short: 81 lines. Most of the development consisted in building up a small hierarchy of meta-theorems with trivial tactics to support them, and then writing one large tactic that built the central part of the proof automatically.

The normal form theorem and reflection

These two are closely related, since most of the work that was done for the normal form theorem could easily be carried over for use with reflection. Since this was the first work done, there was a lot more emphasis on getting things done with tactics instead of meta-theorems. While this made things slower than it would otherwise have been, it also meant that almost all of the development work for the prenex form theorem was directly useable with the reflection work. If greater emphasis had been put on developing useful metatheorems rather than tactics then a lot of theorems would have had to be reproven almost from scratch (this is an example of one of the larger holes in the system: it would in fact have been possible to use the lemmas from the old system in the new system, even though they had not reproved for the new definitions, because they have the same form, even though the underlying definitions have changed).

The lengths of the definitions of the original and the modified systems are about the same. The definition itself is split into several parts: the language of sorted predicate logic, the sequent calculus, and the definition of the rules and axioms. In total, about 1200 lines (in fact because the specification for the reflective form of the system was written much later, and we were able to make use of many extra features

of the compilers to tidy up the original in places, it is actually about 50 lines shorter than its ancestor).

It is more difficult than above to say how long the development for the prenex form theorem took, because at the same time as developing the proof the system itself was was continuously being developed and modified. However, we would estimate that a couple of months was taken up by debugging the specification and then building the proof. However rebuilding the proof (on a responsive machine) took about a working week, given all the various tactics in place. The central proof and the function take a day of hard work each (this is partly because, even on a fast machine, the system is slow enough to break up the user's concentration).

The reflection examples, unlike the prenex form in that they are object level deductions that do not involve any complicated meta-level work (i.e., inductions over proofs or formulae), go through very quickly, once the machinery for the theory has been built.

The lambda calculus

The lambda calculus development took place between the normal form theorem and the deduction theorem. The length of the specification is 170 lines, and took only a couple of hours to write. However developing the function again took several weeks of work, since it was necessary to build new tools to work with the more complicated functions that were needed here than elsewhere.

9.5 Conclusions

While the system is not suitable for real use as a practical framework PDS, this is not because FS_0 itself is unsuitable but because of the implementation, which is an experiment built on top of a system intended for other purposes that here which make it insecure, slow, and sometimes inconvenient. Speed particularly is an issue: even in this implementation, every effort was made to ensure that (e.g.) function evaluation is as fast as possible. This is an argument against an implementation of FS_0 on top of another generic, or framework, theory such as *Isabelle*, since there such facilities would be even more difficult to supply than here[iii].

On the other hand, there is no reason why it should not be possible to build a very efficient implementation easily on top of *Lisp* or *ML* (or even *Prolog*).

§10 The theory SP in FS_0

As an example of FS_0 in practice, at this point I will discuss an encoding of the theory SP, which was disussed in §2.2 of the earlier chapter on meta-level reasoning (which can be referred back to for details). This will be in several parts. First I look at how to encode $\mathcal{L}_{SP}$ in FS_0, then how to describe the theory SP in FS_0, and finally I describe a proof of the deduction theorem.

10.1 The language $\mathcal{L}_{SP}$

The first thing that has to be done is to define some distinct constant s-expressions, which I will call 'prop', '$\vee$' and '$\neg$'.

$$\text{'prop'} \triangleq O$$
$$\text{'}\vee\text{'} \triangleq (\text{'prop'}, O)$$
$$\text{'}\neg\text{'} \triangleq (\text{'}\vee\text{'}, O)$$

Then the encoding of the expressions can be done as follows. First, atomic propositions are defined to be of the form

$$(\boldsymbol{S}, \text{'prop'})$$

(which means that any expression in FS_0 can be made into a proposition, simply by labeling it with 'prop'). The encoding of the language SP is

$$(\boldsymbol{A}_1 \vee \boldsymbol{A}_2) \equiv ((\boldsymbol{S}_1, \boldsymbol{S}_2), \text{'}\vee\text{'})$$
$$\neg\boldsymbol{A} \equiv (\boldsymbol{S}, \text{'}\neg\text{'})$$

(where $\boldsymbol{S}, \boldsymbol{S}_1$ and $\boldsymbol{S}_2$ are the encodings of $\boldsymbol{A}, \boldsymbol{A}_1$ and $\boldsymbol{A}_2$). This definition can be given in FS_0 using the $\mathcal{I}_2(\cdot,\cdot)$ constructor. First the base case is defined

$$pr_{SPC} \triangleq \mathsf{tup}\ (x, \text{'prop'})$$

then the step cases are

$$orgen_{SPC} \triangleq \frac{x \qquad y}{((x,y), \text{'}\vee\text{'})}$$
$$neggen_{SPC} \triangleq \frac{x \qquad -}{(x, \text{'}\neg\text{'})}$$

so that the class of encoded terms of $\mathcal{L}_{SP}$ is defined as:

$$lang_{SPC} \triangleq \mathcal{I}_2(pr_{SPC}, orgen_{SPC} \cup neggen_{SPC})$$

10.2 The theory SP

With the language encoded, the theory SP itself can be encoded. SP has the advantage of having only rules dependent on two previous members, so it is not necessary to resort to any extra coding to make them work here in a definition using the $\mathcal{I}_2(\cdot,\cdot)$ rule. The first thing that has to be defined is the base class of axioms

$$\begin{aligned} exmid_{SPC} &\triangleq \mathsf{tup}\ (((x, \text{`}\neg\text{'}), x), \text{`}\vee\text{'}) \\ &\quad \mathsf{where}\ x \in lang_{SPC} \end{aligned}$$

then the encodings of the rules are

$$\begin{aligned} expand_{SPC} &\triangleq \frac{x \qquad -}{((y, x), \text{`}\vee\text{'})} \\ &\quad \mathsf{where}\ y \in lang_{SPC} \\ contract_{SPC} &\triangleq \frac{((x, x), \text{`}\vee\text{'}) \qquad -}{x} \\ associate_{SPC} &\triangleq \frac{((x, ((y, z), \text{`}\vee\text{'})), \text{`}\vee\text{'}) \qquad -}{((((x, y), \text{`}\vee\text{'}), z), \text{`}\vee\text{'})} \\ cut_{SPC} &\triangleq \frac{(((x, \text{`}\neg\text{'}), y), \text{`}\vee\text{'}) \qquad ((x, z), \text{`}\vee\text{'})}{((y, z), \text{`}\vee\text{'})} \end{aligned}$$

then the encoding of the theory T in FS_0 is

$$\begin{aligned} rules_{SPC} &\triangleq expand_{SPC} \cup contract_{SPC} \cup associate_{SPC} \cup cut_{SPC} \\ T &\triangleq \mathcal{I}_2(exmid_{SPC}, rules_{SPC}) \end{aligned}$$

and it is then possible to associate the encoding with the original theory[iv] as

$$\vdash_{SP} \boldsymbol{A} \qquad \textit{iff} \qquad \vdash_{FS_0} \boldsymbol{S} \in T$$

(where $\boldsymbol{A}$ is encoded as $\boldsymbol{S}$).

10.3 Basic tactics, etc.

Now I have a formal description of the theory SP, but not in a very useable state. The next thing that has to be done is the construction of a collection of meta-level tactics that allow me to work more easily in the theory. The first of these that should be constructed are some that will allow me to apply rules directly. For example I might want a set of tactics that allow me to reason in a 'top down' fashion; in other words, so that given a goal of the form

$$\vdash_{FS_0} \boldsymbol{S} \in T$$

and a rule instance

$$\frac{\boldsymbol{S_2}}{\boldsymbol{S_1}}$$

the appropriate tactic can then be called, to perform a reduction equivalent to

$$\vdash_{FS_0} \boldsymbol{S_1} \in T \qquad \textit{if} \qquad \vdash_{FS_0} \boldsymbol{S_2} \in T$$

without having to go through the intermediate stages manually.

In fact the components of these tactics themselves consist mostly of meta-theory. Consider as an example the tactic for $contract_{SPC}$; it would be easy to write a tactic which performed all the primitive reductions necessary to get the effect of

$$\vdash_{FS_0} \boldsymbol{S_1} \in T \qquad \textit{if} \qquad \vdash_{FS_0} (\boldsymbol{S_1} \vee \boldsymbol{S_1}) \in T$$

but it is much more efficient first to prove the lemma

$$\forall x(((x,x),\text{`}\vee\text{'}) \in T \rightarrow x \in T);$$

so that all the actual tactic, as implemented, has to do is cut in the lemma and instantiate its universal quantifier. Then, if a top down proof is being constructed, the consequent of the instance of the lemma will be the goal, so by eliminating the implication in the lemma, the goal can be reduced to the antecedent; if, on the other hand, a bottom up proof is being constructed, the antecedent will already be on the hypothesis list, and the lemma can again be eliminated, only this time leaving the consequent on the hypothesis list, and the goal untouched. Notice that the rules $exmid_{SPC}$ and $expand_{SPC}$ both have side conditions on them, requiring that,

respectively, $\boldsymbol{A}$, and, $\boldsymbol{B}$ be shown to be well formed, i.e., members of $lang_{SPC}$. This does not produce difficulties when dealing with ground terms; a tactic to do the work is easy to construct, however there are cases where it is a problem, as will be seen.

Five tactics in all have to be built initially: `expand`, `contract`, `associate`, `cut` and `exmid`. But these can be expanded with others. The first obvious extension is commutativity, and this can be implemented in the same way as the basic tactics that have just been defined, using a lemma. The form that the lemma should take is a problem though. The simplest form has extra information in the antecedent, so that the statement is

$$\forall x \forall y(((x,y),\text{`}\vee\text{'}) \in T \rightarrow x \in lang_{SPC} \rightarrow ((y,x),\text{`}\vee\text{'}) \in T).$$

But the explicit statement that x is in $lang_{SPC}$ was not in the original statement of the lemma, and, intuitively is not needed. Remove this extra information though, and the problem of proving it separately arises. If this was just a simple deduction, then there would be no problem, since x would be a ground term, and it would be possible to apply a tactic, like was mentioned above, to prove it is a well formed term. Unfortunately though, here a tactic will not do, since x is simply a free variable; something the tactic cannot cope with. Instead it is necessary to do some further meta-theory; two lemmas have to be proven:

$$T \subset lang_{SPC}$$

and

$$lang_{SPC} \subset \boldsymbol{C}$$

where

$$\begin{aligned} x \in \boldsymbol{C} \leftrightarrow\ & \pi_2 x = \text{`prop'} \vee \\ & (\pi_2 x = \text{`}\neg\text{'} \wedge \pi_1 x \in lang_{SPC}) \vee \\ & (\pi_2 x = \text{`}\vee\text{'} \wedge \pi_{11} \in lang_{SPC} \wedge \pi_{21} \in lang_{SPC}), \end{aligned}$$

which both follow easily from applications of $ClassInd$. After these results have been obtained, it is easy to show that, since $\vdash_{FS_0} ((x,y),\text{`}\vee\text{'}) \in T$, it follows that $\vdash_{FS_0} x \in lang_{SPC}$. The effort of proving these two lemmas is not wasted either, since they are ubiquitous in proofs of meta-theorems about SP, as it is here defined.

10.4 The deduction theorem

The final theorem discussed in the chapter about meta-level reasoning was the deduction theorem. I am now in a position to present a proof of it for the FS_0 implementation I have just described. Consider the informal statement again:

$$\vdash_T \neg A \vee B \qquad \textit{iff} \qquad \vdash_{T[A]} B$$

This is an if and only if statement, but in one direction it is quite trivial to prove. What is interesting, both as a proof in itself, and a meta-theorem that can be used to aid real work, is the right to left implication. A simple formalisation of this in FS_0, for the theory SP would be

$$\begin{array}{l}\forall x\,(x \in lang_{SPC} \rightarrow \\ \quad \forall y\,(y \in \mathcal{I}_2(exmid_{SPC} \cup \{x\}, rules_{SPC}) \rightarrow \\ \quad\quad (((x, \text{`}\neg\text{'}), y), \text{`}\vee\text{'}) \in \mathcal{I}_2(exmid_{SPC}, rules_{SPC}))).\end{array}$$

Notice that this insists that whatever is added as a new axiom is actually a statement in the language. This formulation states the deduction theorem to be an admissible rule of SP. However, it is possible state it in the more general form of a derived rule, which is correspondingly more useful. First a new (schematic) class is defined:

$$T'\langle \boldsymbol{C} \rangle \triangleq \mathcal{I}_2(exmid_{SPC} \cup (lang_{SPC} \cap \boldsymbol{C}), rules_{SPC});$$

then the deduction theorem for SP can be stated as

$$\begin{array}{l}\forall C \forall x\,(x \in lang_{SPC} \rightarrow \\ \quad \forall y\,(y \in T'\langle C \cup \{x\}\rangle \rightarrow \\ \quad\quad (((x, \text{`}\neg\text{'}), y), \text{`}\vee\text{'}) \in T'\langle C\rangle)).\end{array}$$

As is, this is not directly provable, but it follows directly from

$$\begin{array}{l}\forall C \forall x\,(x \in lang_{SPC} \rightarrow \\ \quad T'\langle C \cup \{x\}\rangle \subset \mathcal{P}[[K_x, K_{\text{`}\neg\text{'}}], I, K_{\text{`}\vee\text{'}}]^{-1} T'\langle C\rangle)\end{array}$$

And this follows, after stripping off the two outside universal quantifiers, by structural induction (i.e., applying the rule *ClassInd*).

Before outlining the proof itself though, I should say that it is heavily automated, depending crucially on an object-level tactic `subdis` (for 'sub-disjunction'), which needs to be explained. Given a goal

$$\vdash_{FS_0} ((\boldsymbol{S_1}, \cdots ((\boldsymbol{S_{m-1}}, \boldsymbol{S_m}), \text{`}\vee\text{'}) \cdots), \text{`}\vee\text{'}) \in T$$

which the user wants to reduce to a goal of the form

$$\vdash_{FS_0} ((\boldsymbol{S}_{i_1}, \cdots ((\boldsymbol{S}_{i_{m-1}}, \boldsymbol{S}_{i_m}), \text{`}\vee\text{'}) \cdots), \text{`}\vee\text{'}) \in T$$

(where if x in $\{\boldsymbol{i}_1, \ldots, \boldsymbol{i}_n\}$ then $1 \leq x \leq n$), `subdis` called with a description of the desired subgoal reduces the current goal to that. The code to do this is not trivial, but neither is it interesting; it is actually a fairly direct implementation of an algorithm given in the proof of a theorem stating something very similar in §3.1 [68] (where details can be found). An important detail that has to be dealt with in constructing the tactic is what to do with the enormous numbers of side conditions that are generated at every invocation of `contract` or `exmid`, when it is necessary to show that something is a well formed formula. Since here the intention is that the tactic be used for work at the meta-level (i.e., dealing with schematic formulae containing meta-variables) the same problem as with the lemma used in `commute` arises: that a tactic will not be able to cope with the non-ground terms. Again, the solution is to make use of the two lemmas originally proven above to assist in proving the commutativity lemma, only this time due to the volume of different goals that will have to be constructed, it is better to write tactics so that the actual work can be carried out automatically.

Now the proof can be presented. Having stripped off the two quantifiers and the implication, the goal is

$$x \in lang_{SPC} \vdash_{FS_0} T'\langle C \cup \{x\}\rangle \subset \mathcal{P}[K_x, K_{\text{`}\neg\text{'}}, I, K_{\text{`}\vee\text{'}}]^{-1} T'\langle C\rangle$$

(In future, for the sake of conciseness, $x \in lang_{SPC}$ will not be shown in the hypothesis list, but taken as implicit). On application of *ClassInd* this goal reduces to two subgoals: a base case

$$w \in exmid_{SPC} \cup ((C \cup \{x\}) \cap lang_{SPC}) \vdash_{FS_0} w \in \mathcal{P}[[K_x, K_{\text{`}\neg\text{'}}], I, K_{\text{`}\vee\text{'}}]^{-1} T'\langle C\rangle \qquad (DT_B)$$

and a step case

$$\begin{aligned} & y \in \mathcal{P}[K_x, K_{\text{`}\neg\text{'}}, I, K_{\text{`}\vee\text{'}}]^{-1} T'\langle C\rangle, \\ & z \in \mathcal{P}[K_x, K_{\text{`}\neg\text{'}}, I, K_{\text{`}\vee\text{'}}]^{-1} T'\langle C\rangle, \\ & (w, y, z) \in rules_{SPC} \\ & \qquad \vdash_{FS_0} w \in \mathcal{P}[K_x, K_{\text{`}\neg\text{'}}, I, K_{\text{`}\vee\text{'}}]^{-1} T'\langle C\rangle \qquad (DT_S) \end{aligned}$$

The base case

Take the base case DT_B first. This can be simplified to

$$w \in exmid_{SPC} \cup ((\boldsymbol{C}_1 \cup \{x\}) \cap lang_{SPC}) \vdash (((x, \text{`}\neg\text{'}), w), \text{`}\vee\text{'}) \in T'\langle C\rangle$$

and then the context can be broken up into three subcases

$$w \in exmid_{SPC} \vdash_{FS_0} (((x, \text{`}\neg\text{'}), w), \text{`}\vee\text{'}) \in T'\langle C\rangle \qquad (B_1)$$

$$w \in C,\ w \in lang_{SPC} \vdash_{FS_0} (((x, \text{`}\neg\text{'}), w), \text{`}\vee\text{'}) \in T'\langle C\rangle \qquad (B_2)$$

$$w \in \{x\},\ w \in lang_{SPC} \vdash_{FS_0} (((x, \text{`}\neg\text{'}), w), \text{`}\vee\text{'}) \in T'\langle C\rangle \qquad (B_3)$$

The first case, B_1 follows, since if w is in $exmid_{SPC}$ then it is in $T'\langle C\rangle$. By this, we have

$$w \in T'\langle C\rangle,\ x \in lang_{SPC} \vdash_{FS_0} (((x, \text{`}\neg\text{'}), w), \text{`}\vee\text{'}) \in T'\langle C\rangle$$

and an application of *expand* is enough to finish off the goal.

The second case B_2 follows in almost exactly the same way as the first. B_3, on the other hand, demands a different technique. Since $w \in \{x\}$, $w = x$, so the goal reduces to

$$\vdash_{FS_0} (((x, \text{`}\neg\text{'}), x), \text{`}\vee\text{'}) \in T'\langle C\rangle$$

Which follows immediately, since this is an axiom, and a member of $exmid_{SPC}$.

The step case

Again, the first step is to simplify the goal DT_S to

$$\begin{aligned}
&(((x, \text{`}\neg\text{'}), y), \text{`}\vee\text{'}) \in T'\langle C\rangle,\\
&(((x, \text{`}\neg\text{'}), z), \text{`}\vee\text{'}) \in T'\langle C\rangle,\\
&(w, y, z) \in rules_{SPC}\\
&\quad \vdash_{FS_0} (((x, \text{`}\neg\text{'}), w), \text{`}\vee\text{'}) \in T'\langle C\rangle
\end{aligned}$$

This goal too can be reduced (by expanding the definition of $rules_{SPC}$, and some judicious thinning) to four subcases. They are all of the form

$$\begin{aligned}
&(((x, \text{`}\neg\text{'}), y), \text{`}\vee\text{'}) \in T'\langle C\rangle,\\
&(((x, \text{`}\neg\text{'}), z), \text{`}\vee\text{'}) \in T'\langle C\rangle,\\
&(w, y, z) \in associate_{SPC}\\
&\quad \vdash_{FS_0} (((x, \text{`}\neg\text{'}), w), \text{`}\vee\text{'}) \in T'\langle C\rangle
\end{aligned}$$

except that $associate_{SPC}$ is replaced with cut_{SPC}, $contract_{SPC}$ or $expand_{SPC}$.

The proofs of these, using the `subdis` tactic, are quite similar. The example here can be paraphrased as:

'Show that, if $(((x,\text{`}\neg\text{'}),y),\text{`}\vee\text{'}) \in T'\langle C\rangle$, and $(((x,\text{`}\neg\text{'}),z),\text{`}\vee\text{'}) \in T'\langle C\rangle$ and w follows from y,z by *associate*, then $(((x,\text{`}\neg\text{'}),w),\text{`}\vee\text{'}) \in T'\langle C\rangle$.'

If the details of the relationship between w and y,z are expanded, this goal becomes, formally (since, for instance, z is seen to be not relevant)

$$(((x,\text{`}\neg\text{'}),((\pi_{1111}w,((\pi_{2111}w,\pi_2 w),\text{`}\vee\text{'})),\text{`}\vee\text{'})),\text{`}\vee\text{'}) \in T'\langle C\rangle$$
$$\vdash_{FS_0} (((x,\text{`}\neg\text{'}),((((\pi_{1111}w,\pi_{2111}w),\text{`}\vee\text{'}),\pi_{21}w),\text{`}\vee\text{'})),\text{`}\vee\text{'}) \in T'\langle C\rangle$$

Moving away from the PDS for a moment, given the equivalences $x = A_1$, $\pi_{1111}w = A_2$, $\pi_{2111}w = A_3$ and $\pi_{21}w = A_4$ it is possible plan a proof for this as follows:

$$\dfrac{\dfrac{\dfrac{\vdash_{FS_0} (\neg A_1 \vee (A_2 \vee (A_3 \vee A_4)))}{\vdots\ \texttt{subdis}}}{\vdash_{FS_0} (A_2 \vee (A_3 \vee (A_4 \vee \neg A_1)))}\ associate}{\vdash_{FS_0} ((A_2 \vee A_3) \vee (A_4 \vee \neg A_1))}\ associate$$
$$\dfrac{\vdash_{FS_0} (((A_2 \vee A_3) \vee A_4) \vee \neg A_1)}{\vdots\ \texttt{commute}}$$
$$\vdash_{FS_0} (\neg A_1 \vee ((A_2 \vee A_3) \vee A_4))$$

With this proof-in-SP in hand, the proof can be constructed automatically on returning to the PDS again. And the other cases are dealt with in pretty much the same manner. With this result in hand as a lemma of FS_0 it can then be invoked as a lemma, instantiated and reduced, at no more cost than a basic rule[v].

§11 Summary and conclusions

This has been a long chapter. Feferman's theory FS_0 has been discussed, along with some of its more important properties. Also the tools that have been implemented in order to improve its usability have been mentioned. And finally a small example, of an encoding of a system of propositional logic, has been presented.

In the light of this it is possible to say that certainly FS_0 is effective as a framework for working with a very simple logic such as SP, and that it supplies sufficient facilities that it is possible to prove and use meta-theorems to make the system much more flexible. However, SP is an artificial example, in that it is unlikely

to be encountered in practice. The next chapter considers the problem of how one would go about encoding a theory that has bound variables, which is a much more demanding test.

Notes

i. However, the availability of larger classes would make it possible to work with semi-formal theories, e.g., using an ω-rule; but this possibility is not investigated here.

ii. The 'meta-level' facilities in *Prolog* should not be confused with the idea of meta-theory here. The two have much in common (in fact, the meta-level facilities in *Prolog* even offer something like the reflection facilities discussed later, but they are distinctive not so much for allowing safe formally analysable extensions to the theory, as for allowing unsafe and unanalysable extensions.

iii. Another argument against implementing FS_0 on a framework system is of course simply that it is a framework itself, so by the time it is being used for work in a declared logic the user has to work through two levels of implementation: the framework, and FS_0 itself.

iv. There are issues of faithfulness and adequacy that there is not room to address here.

v. It is worth pointing out that the tactic `subdis` consumes an enormous amount of time: it is used six or seven times in this derivation, and each time it is used it generates several tens, to, sometimes, well over a hundred proof steps. If it really were used in a tactic that aped the deduction theorem, the derivation, which is after all trivial, would consume impossibly large amounts of time and space resources.

Declaring a Language and a Theory

There are two parts to the presentation of a logic in a framework; first is the presentation of the set of all the sentences of a language; then a theory of this language can be defined as a subset of this language that is the smallest subset defined as a class of axioms closed under a given set of rules. This chapter describes the way that FS_0 can be used as a framework for declaring a general first order language (and that will be a useful base for theories used in future chapters of this work), along with a general description of how sequent calculus theories can be defined on top of it.

§1 The abstract form of a language

As an example of a typical and flexible language, this section presents a notation that is suitable for first order sorted logic. Not only does this give an idea of how FS_0 works on a real logic, but it also describes a language that will be useful in the rest of this thesis.

Having settled on the language of sorted predicate logic, the first thing to do is give a description of it, and such a description breaks up into two parts: the language of terms and the language of formulae, each of which can be dealt with separately.

1.1 Conventions

The same conventions are adopted here as in the last chapter, where possible. One addition is that s-expression constants in FS_0 are sometimes represented as quoted strings, i.e., 'constant'. $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ vary over formulae in the language to be encoded, $\boldsymbol{v}$ varies over variables in the language to be encoded, $\boldsymbol{t}$ varies over terms in the language to be encoded, $\boldsymbol{Q}$ varies over quantifiers, and $\Gamma, \Delta, \mathrm{E}$ vary over sets of hypotheses.

1.2 The language of terms $\mathcal{L}_t$

- Variables are terms. Each variable has a sort $\boldsymbol{s}$ and an index $\boldsymbol{i}$ which together uniquely define it, so that ${}_{i}v_{s}$ is the $\boldsymbol{i}$th of sort $\boldsymbol{s}$ (this does not preclude there also being variables ${}_{i'}v_{s}$ and ${}_{i}v_{s'}$). $\boldsymbol{s}$ and $\boldsymbol{i}$ are not further defined except as varying over some countable sets.
- Function applications are terms. A function has a sort $\boldsymbol{s}$, an index $\boldsymbol{i}$ and a parameter sort list $\boldsymbol{p}$ which uniquely define it. A function application is written ${}_{i}f_{s}^{\boldsymbol{p}}\boldsymbol{l}^*$, where $\boldsymbol{l}^*$ is a list of terms and $\boldsymbol{p}$ and $\boldsymbol{l}^*$ agree. Agreement is defined as follows:
  - If $\boldsymbol{p}$ is the empty list then it agrees with $\boldsymbol{l}^*$ if and only if $\boldsymbol{l}^*$ is an empty list.
  - If $\boldsymbol{p}$ is not the empty list then $\boldsymbol{p}$ agrees with $\boldsymbol{l}^*$ if and only if the head of $\boldsymbol{p}$ is the sort of the head of $\boldsymbol{l}^*$ and the tail of $\boldsymbol{p}$ agrees with the tail of $\boldsymbol{l}^*$.
- Nothing else is a term^{i}.

1.3 The language of formulae $\mathcal{L}$

- Atomic predicate applications are formulae. A predicate has an index $\boldsymbol{i}$ and a argument sort list $\boldsymbol{p}$ which uniquely define it, and a predicate application is written ${}_{i}p^{\boldsymbol{p}}\boldsymbol{l}^*$, where $\boldsymbol{l}^*$ is a list of terms and $\boldsymbol{p}$ and $\boldsymbol{l}^*$ agree.
- $\bot$ is a formula.
- If $\boldsymbol{A}$ and $\boldsymbol{B}$ are formulae, then $(\boldsymbol{A} \circ \boldsymbol{B})$ is a formula, where $\circ$ is one of $\vee$, $\wedge$ or $\rightarrow$.
- If $\boldsymbol{A}$ is a formula and $\boldsymbol{v}$ is a variable then $\boldsymbol{Q}\boldsymbol{v}\boldsymbol{A}$ is a formula, where $\boldsymbol{Q}$ is either $\exists$ or $\forall$.
- Nothing else is a formula.

1.4 Commentary

The first thing to notice is that there is no negation sign; instead, there is the symbol $\perp$, which can be used to fulfil the same purpose (that is, $\boldsymbol{A} \rightarrow \perp$ instead of $\neg \boldsymbol{A}$). This is probably more common in constructive mathematics, but is also sometimes used in descriptions of classical systems. Also it is fair criticism to remark that this is not a very readable language and, as will be seen, the way it is actually written in FS_0 it is even more unreadable. But though this is the form of the language that the machine has to deal with, it is not to be expected that the user actually see it in this form. The implementation should (and does) provide ways to hide this messiness from the user.

§2 The syntax of $\mathcal{L}$ in FS_0

The description of $\mathcal{L}$ given above is not enough to allow its implementation in FS_0; before that can be done, a mapping to s-expressions has to be defined. There is a great deal of flexibility in what is possible, but at the same time the representation is unlikely to be very readable by a person. The language is divided into terms and formulae, and in top down fashion formulae will be treated first (because they are simpler).

The definition of $\mathcal{L}$ distinguishes four types of formulae: atomic predicates, absurdity, propositional formulae, and predicate formulae. These are all represented by s-expressions which are distinguished by constants on the left (essentially trees with nodes which consist of labels on the left and branches on the right). So with the appropriate distinct constants defined a translation (indicated by $\widetilde{\cdot}$) can be defined

$$\begin{aligned}
\widetilde{\perp} &\equiv (O, \text{`}\perp\text{'}) \\
\widetilde{\boldsymbol{P}} &\equiv (\rho, \text{`apred'}) \\
\widetilde{\boldsymbol{A} \circ \boldsymbol{B}} &\equiv ((\widetilde{\boldsymbol{A}}, \widetilde{\boldsymbol{B}}), \text{`}\circ\text{'}) \qquad \circ \in \{\rightarrow, \wedge, \vee\} \\
\widetilde{\boldsymbol{Q}\boldsymbol{v}\boldsymbol{A}} &\equiv ((\widetilde{\boldsymbol{A}}, \tilde{\boldsymbol{v}}), \text{`}\boldsymbol{Q}\text{'}) \qquad \boldsymbol{Q} \in \{\exists, \forall\}
\end{aligned}$$

where $\boldsymbol{v}$ is a variable, $\boldsymbol{P}$ is an atomic predicate ρ is some way of encoding the structure of $\boldsymbol{P}$ (defined below), $\boldsymbol{A}$ and $\boldsymbol{B}$ are formulae, and '$\perp$','$\rightarrow$', etc. are defined to be distinct constants. Notice that the the absurd formula is not a 'leaf,' so much as a tree branch with no descendents which is labeled with the absurd constant.

The structure of terms (and the internals of atomic predicates, which will be grouped with terms here) is more complex, because when dealing with terms it is necessary to take account of the sort, and possibly the index and the sorts of the

arguments as well. The same idea of a tree labeled on the left still holds though, only the structure of the label is more complex.

The general form of a term is:

$$(\cdot_1, (\cdot_2, \cdot_3))$$

where $\cdot_3$ contains a constant label, $\cdot_2$ contains information on the index, the sort of the term, and the sorts of the arguments, and $\cdot_1$ contains the argument list. For a variable it becomes simply:

$$\widetilde{\widetilde{\widetilde{\widetilde{{}_i v_s}}}} \equiv (O, ((\tilde{s}, \tilde{i}), \text{'var'}))$$

where s varies over the sorts, and i varies over the indexes, and O is the empty argument list (since variables do not have arguments). We do not define further the sorts or the index, simply requiring that there is some way to map them into the s-expressions, so that they can be encoded. For functions and atomic predicate applications this becomes, not quite so simply:

$$\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{{}_i f_s^{\boldsymbol{p}} \boldsymbol{l}^*}}}}} \equiv (\widetilde{\widetilde{\boldsymbol{l}^*}}, ((\tilde{s}, (\tilde{\boldsymbol{p}}, \tilde{\boldsymbol{i}})), \text{'fn'}))$$
$$\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{{}_i p^{\boldsymbol{p}} \boldsymbol{l}^*}}}}} \equiv (\widetilde{\widetilde{\boldsymbol{l}^*}}, ((\tilde{\boldsymbol{p}}, \tilde{\boldsymbol{i}}), \text{'pr'}))$$

where s and i are as before, while l^* is a list of arguments and p is a list of argument sorts. Notice that the position of s is the same for both function applications and variables (at π_{112}), so that it is easy to check a list of terms against a argument typing without having to bother about distinguishing the two (this is simply to make implementation easier).

§3 Implementing $\mathcal{L}$ in FS_0

Now, finally, it is possible to give a definition in FS_0 of $\mathcal{L}$. The definition below is described top down, though in fact it has to be given to the machine bottom up.

3.1 Defining the formulae

First there are the two base classes of formulae, the atomic predicates and the class containing only the void sentence. Definition of the former is deferred to the next section (along with var_C), so for the moment it will be left simply as $rpred_C$, the other is defined simply as:

$$absurds_C \triangleq [I, K_{(O,\text{`}\perp\text{'})}]^{-1}\, equal_C$$

i.e., the class containing only the sentence 'absurd'. Notice that this is not the symbol '$\perp$', which is a connective (albeit a nullary one), but rather a tree node labeled with '$\perp$' which has no subtrees.

On examining the recursive definitions for the formulae one can see that no possibility is dependent on more than two other formulae for its definition; this is convenient since it means that I can make direct use of the $\mathcal{I}_2(\cdot,\cdot)$ constructor for classes. But before this, the definitions of $propcon_C$ and $predcon_C$ (the binary connectives and the quantifiers) have to be supplied; which is trivial

$$\begin{aligned}
conj_C &\triangleq [I, K_{\text{`}\wedge\text{'}}]^{-1}\, equal_C \\
&\vdots \\
forall_C &\triangleq [I, K_{\text{`}\forall\text{'}}]^{-1}\, equal_C \\
propcon_C &\triangleq conj_C \cup implies_C \cup disj_C \\
predcon_C &\triangleq exists_C \cup forall_C
\end{aligned}$$

Then the relations for the recursive cases are just:

$$propg_C \triangleq \frac{left \qquad right}{((left, right), con)}$$
$$\text{where } con \in propcon_C$$

$$predg_C \triangleq \frac{form \qquad -}{((form, v), q)}$$
$$\text{where } q \in predcon_C$$
$$v \in var_C$$

So that the definition of $\mathcal{L}$ (written wff_C) is simply:

$$wff_C \triangleq \mathcal{I}_2(rpred_C \cup absurds_C,\ propg_C \cup predg_C)$$

3.2 Defining the terms

The definition of the class of terms is more complex than that of the class of formulae. In fact three classes are defined together: the class of terms, the class of lists of terms, and the class of atomic propositions. Each is labeled appropriately, so that the class of terms can be projected out afterwards (as can the class of atomic predicates).

The superclass containing the terms is called $rpredsetc_C$. It is defined in terms of $\mathcal{I}_2(\cdot,\cdot)$ and its members are labeled, as usual, on the right.

The base class is the set containing the empty list and the class of labeled variables. From these it is possible to define the other objects in the class, so they are dealt with first.

$$var_C \triangleq \mathsf{tup}\ (O,((s,i),\text{'var'}))$$

$$bt_C \triangleq \mathsf{tup}\ (v,\text{'term'})$$
$$\mathsf{where}\ v \in var_C$$

$$emptytl_C \triangleq \mathsf{tup}\ (O,\text{'terml'})$$

Then come the various relations for the recursive definitions. The easiest, and the first one dealt with is for extending a list of terms—given a term list, it can be extended by adding another term at the head:

$$extl_C \triangleq \frac{(l,\text{'terml'}) \qquad (t,\text{'term'})}{((l,t),\text{'terml'})}$$

Function and predicate applications are almost the same, except that function applications are typed, a property that predicate applications do not need, and of course they are labeled differently:

$$termg_C \triangleq \frac{(l,\text{'terml'}) \qquad -}{((l,f),\text{'term'})}$$
$$\mathsf{where}\ f \in fnId_C$$
$$(l,\pi_{121},f) \in wt_C$$

$$rpredg_C \triangleq \frac{(l,\text{'terml'}) \qquad -}{((l,p),\text{'rpred'})}$$
$$\mathsf{where}\ p \in prId_C$$
$$(l,\pi_{11},p) \in wt_C$$

(where wt_C is the class defining the relation between the list of sorts of the arguments to a function or predicate, and a list of terms) at which point it is possible to define the class $rpredsetc_C$, and from it the class $rpred_C$ as:

$$\begin{aligned} rpredsetc_C &\triangleq \mathcal{I}_2(bt_C \cup emptytl_C,\ extl_C \cup termg_C \cup rpredg_C) \\ apred_C &\triangleq \mathsf{tup}\ (a, \text{'rpred'}) \\ &\quad \mathsf{where}\ (a, \text{'rpred'}) \in rpredsetc_C \\ term_C &\triangleq \mathsf{tup}\ a \\ &\quad \mathsf{where}\ (a, \text{'term'}) \in rpredsetc_C \end{aligned} .$$

This leaves the classes wt_C, $fnId_C$ and $prId_C$ to be defined. The second two of these are easy (they are just the classes of predicate and function identifiers):

$$fnId_C \triangleq \mathsf{tup}\ ((s,(p,i)), \text{'fn'})$$

$$prId_C \triangleq \mathsf{tup}\ ((p,i), \text{'pr'})$$

Notice that there are no restrictions on the index, the arguments or the typing of either of these. This might seem slightly strange in the case of the argument list, since that is explicitly a list, but (like in Pure Lisp) any s-expression is a list if the π_2 and π_1 are regarded as the head and tail, and O as the null list (since O is the only atom, it is pressed into service to stand for the empty list, which is also the tail).

Checking well-typedness is a matter of extracting the sorts from the list of terms and comparing the resulting list with the argument sort list. To this end a function wtr_F is defined:

$$wtr_F(O) = O$$

$$wtr_F(t,h) = (wtr_F(t), tsort_F(h))$$

Where $tsort_F$ is the function extracting the sort of a term, defined by

$$tsort_F \triangleq \pi_{112}.$$

Thus the definition of wt_C (the well-typedness relation) is:

$$wt_C \triangleq \mathcal{P}[\mathcal{C}[wtr_F, \pi_1], \pi_2]^{-1}\, equal_C$$

§4 Substitution for $\mathcal{L}$

Substitution is an issue that it could be argued is not quite part of the syntax, and not quite part of the theory. Different people have located it differently (e.g., Church, or de Bruijn). But wherever it is located, it is an important and substantial issue. First, the history of the problem shows that it has to be treated carefully (even Hilbert and Ackermann together are reputed to have got the definitions wrong in a book they once wrote together), even now that issues are well understood. Also, in the particular circumstances here, the usual treatments are not quite satisfactory.

4.1 The usual treatments

A good discussion of the issues, and a typical way of dealing with them, is provided by Barendregt in chapter 2.1 of [6]. The central problem is with confusion between the bound variables of the term to be substituted into and the free variables of the term that is substituted in. In other, less abstract, words, a formal definition of substitution which rules out the following is needed:

$$(p(v) \vee \exists v'\, q(v, v'))\,[f(v')/v] \rightsquigarrow p(f(v')) \vee \exists v'\, q(f(v'), v')$$

Here the bound v' on the right of the initial disjunct should have been renamed to a new variable; it is not the same as the instance of v' in $f(v')$. This is called 'confusion of variables': the bound v' has accidentally 'captured' the v' in $f(v')$.

INTERLUDE: *α-congruence.* Before going any further, the concept of α-congruence needs to be explained. First, from §2.1.11 of [6]:

> 'A *change of bound variable* in an expression is the replacement of a part $\boldsymbol{Qv}\boldsymbol{A}$ of $\boldsymbol{B}$ by $\boldsymbol{Qv'}\boldsymbol{A}[\boldsymbol{v'}/\boldsymbol{v}]$, where $\boldsymbol{v'}$ does not occur (at all) in $\boldsymbol{A}$'

Then two terms are α-congruent if one can be constructed, by a series of changes of variable, from the other.

The approach that Barendregt takes in [6] is simply to suppress the problem at the beginning, by declaring that two α-congruent terms are identified at the syntactic level, so $\boldsymbol{Qv}\boldsymbol{P}(v)$ is syntactically *identified* with $\boldsymbol{Qv'}\boldsymbol{P}(v')$. Further, he insists that in any expression the names of the bound variables are chosen to be distinct from the names of the free variables.

In contrast with this, Church [14], places α-convertibility firmly in the theory, having an explicit α-conversion rule. This means that during a substitution operation

it might be necessary to perform some changes of bound variable in the background. It is the treatment of these changes of bound variable that have traditionally produced the problems.

Yet another approach, used in practice in computer science, is that suggested by de Bruijn [11], of nameless variables. Instead of having a binding operator bind a variable name that can be used in the expression, each binding operator marks a scope boundary. Then variables are indicated simply by having them hold the number of nested scopes that separate them from their bindings. This approach is favoured in computer science, since it allows very efficient substitution algorithms (mostly because it removes any need to take account of α-congruence or α-convertibility at all), but has the problem that it is not very effective in dealing with open terms: contexts must be carried around so that free variables can be distinguished. One way to avoid this problem is to have a separate syntactic class of unbound variables, with a different method of dealing with substitution. However, then we would have to reason in parallel about two separate binding mechanisms, which can only complicate matters.

4.2 Problems with the usual treatments

For what is required here, unfortunately, the usual treatments are not quite enough. Neither Barendregt's own approach, nor those of the others that he outlines, take any account of intentional information in the bound variables that I would like to keep here. Consider for instance the general shape of a program specification:

$$\forall input\,(pre\_cond(input) \rightarrow \exists output\,(post\_cond(input, output)))$$

Barendregt identifies α-congruent terms, so this can be thought of as:

$$\forall *\ (pre\_cond(*) \rightarrow \exists *'\ post\_cond(*, *'))$$

Where the only thing that can be said about $*$ and $*'$ is that they are not the same. This is sometimes known as an abstract syntax; *abstract* because it differs from the usual idea, that syntax is strings of symbols. However now, inconveniently and unintuitively, it is not necessarily the case that $\boldsymbol{A}$ is represented (in an implementation) by exactly the same object as $\boldsymbol{A}[v/v]$. Imagine:

$$\begin{aligned}(p(y) \wedge \forall v\, q(v))\,[v/v] &\rightsquigarrow (p(y) \wedge \forall v'\, q(v'))\,[v/v] \\ &\rightsquigarrow p(y) \wedge \forall v'\, q(v')\end{aligned}$$

then it follows that

$$`p(y) \wedge \forall v\, q(v)' \neq `p(y) \wedge \forall v'\, q(v')'.$$

De Bruijn's approach to formalising the the specification might be written as:

$$\exists \cdot (pre\_cond(\cdot_0) \rightarrow \forall \cdot post\_cond(\cdot_1, \cdot_0))$$

Notice that here two instances of what would normally be thought of as one variable have apparently different 'names', and that no explicit variable is attached to the quantifiers. This is because $\cdot_1$ in the consequent has to pass through one scope boundary to get to its binding, while $\cdot_0$ in the antecedent does not have to pass through any scope boundaries to get to the same binding.

Both of these suffer at least from the loss of the name of the bound variable. The former is easy to reason about and the latter is easy to compute with—neither is really suitable (for people) to reason with (users find the information that the name of the quantifier contains very useful to tell them where they are). For this reason a definition of substitution with which, where possible, names are retained as they are, would be preferable, and the following section supplies one.

4.3 Implementation

With this problem in mind, the substitution mechanism is not difficult to specify as a four place relation $subin_C$ of all instances of the schema:

$$(\boldsymbol{A}[\boldsymbol{t}/\boldsymbol{v}], \boldsymbol{A}, \boldsymbol{t}, \boldsymbol{v})$$

The top level of the definition is simply

$$\begin{aligned} subin_C \triangleq\ & \mathsf{tup}\ (a', a, t, v) \\ & \mathsf{where}\ a' = sub_F(t, v, a) \end{aligned}$$

A functional definition like this has useful properties especially from the perspective of meta-theory; specifically it means that there is guaranteed to be a unique value for any particular substitution $a[t/v]$.

The rest of this subsection describes how to define sub_F.

Naïve substitution

The first thing to do is define a function that simply substitutes all occurrences of v with t in a, without worrying about capture. The $naisub_F$ function is the solution of the set of equations

$$naisub_F(t,v,(O,\text{`}\bot\text{'})) = (O,\text{`}\bot\text{'})$$

$$naisub_F(t,v,((a,b),\text{`}\vee\text{'})) = ((naisub_F(t,v,a),naisub_F(t,v,b)),\text{`}\vee\text{'})$$

$$naisub_F(t,v,((a,b),\text{`}\wedge\text{'})) = ((naisub_F(t,v,a),naisub_F(t,v,b)),\text{`}\wedge\text{'})$$

$$naisub_F(t,v,((a,b),\text{`}\rightarrow\text{'})) = ((naisub_F(t,v,a),naisub_F(t,v,b)),\text{`}\rightarrow\text{'})$$

$$naisub_F(t,v,(p,\text{`rpred'})) = (naisub_F(t,v,p),\text{`rpred'})$$

$$naisub_F(t,v,(a,v',\text{`}\exists\text{'})) = \begin{cases} (a,v',\text{`}\exists\text{'}) & \text{if } v = v' \\ (naisub_F(t,v,a),v',\text{`}\exists\text{'}) & \text{if } v \neq v' \end{cases}$$

$$naisub_F(t,v,(a,v',\text{`}\forall\text{'})) = \begin{cases} (a,v',\text{`}\forall\text{'}) & \text{if } v = v' \\ (naisub_F(t,v,a),v',\text{`}\forall\text{'}) & \text{if } v \neq v' \end{cases}$$

$$naisub_F(t,v,(O,(vi,\text{`var'}))) = \begin{cases} t & \text{if } v = (O,(vi,\text{`var'})) \\ (O,(vi,\text{`var'})) & \text{if } v \neq (O,(vi,\text{`var'})) \end{cases}$$

$$naisub_F(t,v,(l,(fi,\text{`fn'}))) = (naisublist_F(t,v,l),(fi,\text{`fn'}))$$

$$naisub_F(t,v,(l,(pi,\text{`pr'}))) = (naisublist_F(t,v,l),(pi,\text{`pr'}))$$

$$naisublist_F(t,v,O) = O$$

$$naisublist_F(t,v,(r,f)) = (naisublist_F(t,v,r),naisub_F(t,v,f))$$

And these are solvable automatically by the function compiler described in the previous chapter. Now it is possible to prove a basic sort preserving property of naïve substitution:

$$\forall v\,(v \in var_C \rightarrow \forall t\,(t \in term_C \rightarrow \forall w\,(w \in wff_C \rightarrow naisub_F(t,v,a) \in wff_C)))$$

which follows by induction on the structure of w and the lemma

$$\begin{aligned} &\forall v\,(v \in var_C \rightarrow \forall t\,(t \in term_C \rightarrow \forall p\,(p \in rpredsetc_C \rightarrow \\ &\quad((\pi_2 p = \text{`rpred'} \wedge naisub_F(t,v,p) \in rpredsetc_C) \vee \\ &\quad(\pi_2 p = \text{`term'} \wedge (naisub_F(t,v,\pi_1 p),\text{`term'}) \in rpredsetc_C) \vee \\ &\quad(\pi_2 p = \text{`terml'} \wedge (naisublist_F(t,v,\pi_1 p),\text{`terml'}) \in rpredsetc_C))))), \end{aligned}$$

which follows by induction on the structure of $rpredsetc_C$.

Of course, $naisub_F$ is not a sound substitution algorithm, since it does not take account of the possibility of free variables in t being captured by variables of the same name that are bound in a.

Renaming bound variables

As a first step towards rectifying this problem, the next thing defined is a function that renames all bound instances of a variable v to a new variable v' (that is assumed not to occur in the term at all).

$$rename_F(v', v, (a, v'', \text{`}\exists\text{'})) = \begin{cases} (naisub_F(v', v, rename_F(v', v, a)), v', \text{`}\exists\text{'}) & \text{if } v = v'' \\ (rename_F(v', v, a), v'', \text{`}\exists\text{'}) & \text{if } v \neq v'' \end{cases}$$

$$rename_F(v', v, (a, v'', \text{`}\forall\text{'})) = \begin{cases} (naisub_F(v', v, rename_F(v', v, a)), v', \text{`}\forall\text{'}) & \text{if } v = v'' \\ (rename_F(v', v, a), v'', \text{`}\forall\text{'}) & \text{if } v \neq v'' \end{cases}$$

$$rename_F(v', v, (p, \text{`rpred'})) = (p, \text{`rpred'})$$

$$rename_F(v', v, (O, \text{`}\bot\text{'})) = (O, \text{`}\bot\text{'})$$

$$rename_F(v', v, ((a, b), \text{`}\vee\text{'})) = ((rename_F(v', v, a), rename_F(v', v, b)), \text{`}\vee\text{'})$$

$$rename_F(v', v, ((a, b), \text{`}\wedge\text{'})) = ((rename_F(v', v, a), rename_F(v', v, b)), \text{`}\wedge\text{'})$$

$$rename_F(v', v, ((a, b), \text{`}\rightarrow\text{'})) = ((rename_F(v', v, a), rename_F(v', v, b)), \text{`}\rightarrow\text{'})$$

Again, like for $naisub_F$, a basic sort preserving property of the system has to be proven:

$$\forall v\,(v \in var_C \rightarrow \forall v'\,(v' \in var_C \rightarrow \forall w\,(w \in wff_C \rightarrow rename_F(v, v', a) \in wff_C)))$$

which again follows by a simple induction on the structure of wff_C and appeal to the sort preserving properties of $naisub_F$.

Free variables in the term

Another function necessary for the substitution implementation is one that returns the variables occuring in the term. This is easily defined as

$$freevars_F((O, (s, i), \text{`var'})) = (O, (O, ((s, i), \text{`var'})))$$

$$freevars_F((l, f) = freevarslist_F(l)$$

$$freevarslist_F(O) = O$$

$$freevarslist_F((r, h)) = append_F(freevars_F(r), freevarslist_F(h))$$

(notice the dummy variable f in the second equation) and it is easy to show that

$$\forall t\,(t \in term_C \rightarrow freevarslist_F(t) \in List\langle term_C \rangle)$$

(this follows directly from a proof that

$$\begin{aligned}\forall p\,(p \in rpredsetc_C \rightarrow \\ &((\pi_2 p = \text{'term'} \wedge freevars_F(\pi_1 p) \in List\langle term_C\rangle) \vee \\ &(\pi_2 p = \text{'terml'} \wedge freevarslist_F(\pi_1 p) \in List\langle term_C\rangle) \vee \\ &\pi_2 p = \text{'rpred'})\end{aligned}$$

Simplifying the free variable list

In order to cut down the size of the free variable list, and so as to make the substitution algorithm easier to program, more efficient, and have slightly better properties, a function is then defined which removes duplicates from the list of free variables as well as the target variable itself.

$$svl_F(v, O) = O$$

$$svl_F(v,(h,v')) = \begin{cases} svl_F(v,h) & \text{if } v = v' \\ svl_F(v,h) & \text{if } member_F(v',h) \\ (svl_F(v,h),v') & \text{otherwise .} \end{cases}$$

Then the sort preserving property is that

$$\forall C \forall l\,(l \in List\langle C\rangle \rightarrow svl_F(l) \in List\langle C\rangle)$$

An alpha conversion function

A function which alpha converts all bound instances of a variable $(y, \text{'var'})$ in a formula a can be defined as

$$\alpha\text{-}cnv_F(v,a) = rename_F(noi_F(tsort_F v, a), v, a)$$

where $noi_F(s,a)$ is a function (discussed below) which returns a variable of sort s that does not occur in a term a.

The substitution function

It is now possible to construct the substitution function. But before continuing doing this, it is worth introducing a notational convenience which, supplies some of the facilities traditionally associated with programming languages that allow higher order functions.

INTERLUDE: *'Higher order functions'* Higher order functions (or functionals) are not available in FS_0 since it is a strictly second order system. However, it is possible to think of classes of functions that can defined in terms of one schema, (like the equivalent to the cross product defined in the last chapter) and it is possible to prove statements about the schema, quantified over the functions and the classes. This allows the same sorts of facilities that higher order functionals allow such as general theorems about all instantiations of the schema. The functional, 'map' (which is used further on in this work) is given by the definition:

$$map\langle f\rangle(a, O) = O$$

$$map\langle f\rangle(a, (t, h)) = (map\langle f\rangle(a, t), f(a, h))$$

which is solvable for any case where f is already defined. It can, for instance be shown, in FS_0, to have the sort preserving property:

$$\forall A \forall B \forall C \forall f$$
$$(\forall a (a \in A \rightarrow \forall b (b \in B \rightarrow f(a, b) \in C)) \rightarrow$$
$$\forall a (a \in A \rightarrow \forall l (l \in List\langle B\rangle \rightarrow map\langle f\rangle(a, l) \in List\langle C\rangle))))$$

Another example is 'inject,' which is defined:

$$inject\langle f\rangle(a, c, O) = c$$

$$inject\langle f\rangle(a, c, (t, h)) = f(a, h, inject\langle f\rangle(a, c, t))$$

with the sort preserving property:

$$\forall A \forall B \forall C \forall f$$
$$(\forall a (a \in A \rightarrow \forall b (b \in B \rightarrow \forall c (c \in C \rightarrow f(a, b, c) \in C))) \rightarrow$$
$$\forall a (a \in A \rightarrow \forall b (c \in C \rightarrow \forall l (l \in List\langle B\rangle \rightarrow inject\langle f\rangle(a, c, l) \in C))))$$

Other general properties of these schemas can be proven in the same way.

Then a definition of sub_F is as follows.

$$sub_F(t, v, a) = naisub_F(t, v, inject\langle f\rangle(O, a, (svl_F(v, freevars_F(t)))))$$

$$where\ f(d, v, a) = \alpha\text{-}cnv_F(v, a)$$

(notice that the function used to instantiate $inject\langle\cdot\rangle$ takes a dummy parameter d). So sub_F constructs a list of the free variables in t, removes duplicates, using $inject\langle\cdot\rangle$, alpha converts in turn all bound instances of variables in a, and then subsitutes the t for each each unbound instance of the variable in the resulting formula.

The final sort preserving property is then

$$A \in wff_C \rightarrow v \in var_C \rightarrow t \in term_C \rightarrow A[t/v] \in wff_C$$

which follows easily from the properties of $inject\langle\cdot\rangle$.

Properties of substitution

At this point it is possible to verify that the substitution function has the properities that are required it.

The first of these is that the identity substitution is an identity operation:

$$\forall a \forall a' \forall x (a \in wff_C \rightarrow a' \in wff_C \rightarrow x \in var_C \rightarrow (a', a, x, x) \in subin_C \rightarrow a = a')$$

This follows from the facts that

$$\forall a (a \in wff_C \rightarrow \forall v (v \in var_C \rightarrow naisub_F(v, v, a) = a))$$

(by induction on the structure of formulae and terms), and

$$\forall a (a \in wff_C \rightarrow \forall v (v \in var_C \rightarrow sub_F(v, v, a) = naisub_F(v, v, a)))$$

(by 'partial evaluation' of the definition of sub_F).

The second useful fact is that subsitution is unique. This follows trivially from the fact that substitution is defined using a function.

We can also easily define a class of substitution instance relations, using the fact that comprehension for Σ^0_1 formulae is available, i.e.,

$$(a', a, v) \in subinst_C \leftrightarrow \exists t (a', a, t, v) \in subin_C$$

4.4 not-free-in

Finally here is the definition of the class nfi_C which defines the relation of a variable not being free in a formula or list of formulae (with the pairs labeled appropriately. The definition does not itself illuminate any new point, so it is omitted here. It is enough to say that

$$(tl, v, \text{'termlst'}) \in nfi_C$$
$$(t, v, \text{'termvar'}) \in nfi_C$$

if tl is a list of terms or t is a term in which v does not occur free. It is then possible using this to define a function noi_F which has the behaviour that

$$\forall s \forall v \forall w (v \in var_C \rightarrow w \in wff_C \rightarrow naisub_F(noi_F(s, w), v, w) = w)$$

and as a corollary of this, that

$$\forall w \forall s (w \in wff_C \rightarrow$$
$$\forall x ((x = \text{'termlst'} \vee x = \text{'termvar'}) \rightarrow (w, noi_F(s, w), x) \in nfi_C)).$$

§5 The presentation of the theory

With a language in place, the next stage is to present a logical theory for that language. There are several well known ways to present a logic though, and they are not equally suitable for all tasks. In what follows several styles of presentations are considered for their suitability here.

What properties of a presentation influence its suitability for working in, or reasoning about? First, and most importantly, the presentation has to be easy to use; sometimes presentations are designed to be easy to reason about, with no consideration about reasoning in. Having said that, since what is of interest here is meta-level reasoning, a formalism that is easy to reason about is also useful (this is, of course, a requirement that need not apply in general). Finally, and pragmatically, the system should be easy to implement in FS_0.

5.1 Hilbert presentations

Hilbert style presentations are one of the oldest ways of presenting logics, and rely heavily on axiom schemas trying to make do with as few rules as possible; for instance detachment (modus ponens) is the only rule needed to define propositional logic, and classical predicate logic requires only one more. This can be a considerable advantage; a small number of rules simplifies the proof theory considerably in some respects. Another advantage is that perhaps subtle rules are easier to add to the logic — most presentations of modal logics are in a Hilbert style for this reason.

The problem with Hilbert presentations is that they are unuseable in practice. Proofs tend to be unintuitive and roundabout. It is significant that one of the first things generally proven about such systems is the deduction theorem, or else (as is the case with modal logics) a remark to the effect that there is no deduction theorem.

5.2 Natural deduction

Natural deduction is a theory that was designed with the opposite intention to Hilbert presentations above: to be easy to use at the object level. It is no coincidence that the first rule usually derived for Hilbert presentations, the deduction theorem, corresponds to a primitive rule in natural deduction (implication introduction).

Various criticisms can be made of natural deduction at the meta-level though, especially for the $\vee$, $\exists$, $\bot$ fragment[ii]. It would be possible to live with these problems to get the object level ease of use, but I decided against this presentation in the end for pragmatic reasons: the discharge mechanism is quite difficult to implement. And, anyway, why build all this machinery when the next alternative (sequent calculus) comes with equivalent carefully thought out facilities for doing just that?

5.3 Sequent calculus

Sequent calculus seems to offer the best of both the above worlds. Like Natural deduction it is easy to use in practice — successful formal reasoning systems like *Nuprl* show that it is useable in practice. And it has good meta theoretic properties, like Hilbert systems; after all it was developed specifically for proof theory.

A final point about sequent calculus is that it avoids the problems mentioned for natural deduction. There is no 'non-locality' in the application of the rules; the notion of a sequent already supplies explicit facilities for taking care of the hypotheses. For

instance the rule in sequent calculus corresponding to implication introduction in natural deduction is the implication right rule:

$$\frac{\Gamma, \boldsymbol{A} \vdash \boldsymbol{B}}{\Gamma \vdash \boldsymbol{A} \to \boldsymbol{B}}$$

which is pretty much identical to the way implication introduction would actually be defined in an implementation of natural deduction[iii].

§6 The implementation of the theory

Having settled on a particular presentation (sequent calculus) for theories, it is necessary to settle on a representation of that presentation in FS_0.

6.1 Sequents

The general notion of sequent is something of the form[iv]:

$$\Gamma \vdash \Delta \qquad \text{where} \qquad \Gamma, \Delta \in \mathbb{F}(\mathcal{L})$$

(where $\mathbb{F}(\cdot)$ indicates a finite powerset). This could easily be implemented in FS_0 as a pair

$$\begin{aligned} seq_C \triangleq \mathsf{tup}\ (a, b) \\ \mathsf{where}\ a \in List\langle\mathcal{L}\rangle \\ b \in List\langle\mathcal{L}\rangle \end{aligned}$$

but here I will give a definition for sequents that have only one consequent, i.e., of the form:

$$\Gamma \vdash \gamma \qquad \text{where} \qquad \Gamma \in \mathbb{P}(\mathcal{L}), \gamma \in \mathcal{L}$$

where the consequent is a single formula, rather than a set. So sequents are represented simply as lists of formulae, where the head of the list is the consequent, and the tail is the collection of hypotheses. The actual definition of the labeled form is written, slightly redundantly:

$$\begin{aligned} seq_C \triangleq \mathsf{tup}\ (lw, w, \text{'seq'}) \\ \mathsf{where}\ lw \in List\langle\mathcal{L}\rangle \\ w \in \mathcal{L} \end{aligned}$$

6.2 Rules

Rules are relations between lists of sequents and sequents of the form:

$$\frac{\mathrm{A} \vdash \alpha \cdots \Psi \vdash \psi}{\Omega \vdash \omega}$$

where the list of sequents above the line are called the subgoals, and the sequent below is called the goal. The obvious way of representing them is simply as non-empty lists of sequents, where the head of the list is the goal, and the tail is the subgoals, thus

$$\begin{aligned} rule b_C \triangleq \mathsf{tup}\ (sl, s) \\ \mathsf{where}\ sl \in List\langle seq_C \rangle \\ s \in seq_C \end{aligned}$$

is the class of all possible rules, and particular classes of rules $rules_C$ will be defined as subsets of this.

6.3 Proofs

Finally, there is the matter, discussed earlier, in chapter 2, of whether or not to retain the proof object. The spirit of Smullyan's original thesis [73] and Feferman's presentation [25] argue for derivability alone. On the other hand, many modern proof development systems proving systems (such as *Lego* and *Nuprl*) do keep track.

In the example described here I have decided to keep the structure of the proofs in the derivation, since the structure of the proof could be useful in various types of reflected or meta level reasoning. In general then, a formal proof is of the form:

$$\frac{\begin{array}{ccc} \vdots & & \vdots \\ \mathrm{A} \vdash \alpha & \cdots & \Psi \vdash \psi \end{array}}{\Omega \vdash \omega}$$

where

$$\begin{array}{ccc} \vdots & & \vdots \\ A \vdash \alpha, \ldots, & \Psi \vdash \psi & \end{array}$$

are themselves proofs, and

$$((O, \widetilde{\mathrm{A} \vdash \alpha}, \ldots, \widetilde{\Psi \vdash \psi}), \widetilde{\Omega \vdash \omega}) \in rules_C$$

So a proof is actually represented as a pair of a list of proofs on the left and a sequent on the right.

Since a proof is actually constructed out of a list of proofs, the classes of proofs and lists of proofs are defined together in the definition of $prfsEtcl_C$, labeled respectively 'prfl' and 'prf'.

The base class of lists of proofs is simply the class of the tuple containing only the appropriately labeled empty list:

$$eprfl_C \triangleq [i, K_{(O,\text{'prfl'})}]^{-1}\, equal_C$$

and the relation extending it is:

$$prflg_C \triangleq \frac{(l, \text{'prfl'}) \qquad (p, \text{'prf'})}{(l, p, \text{'prfl'})}$$

Proofs can then be built using the definition:

$$prfg_C \triangleq \frac{(pl, \text{'prfl'}) \qquad (p, \text{'prf'})}{(pl, p, \text{'prf'})}$$

$$\text{where } p \in seq_C$$

$$(gleft_F, pl, p) \in rules_C$$

where $gleft_F$ is a function that takes a list of proofs and returns a list of the corresponding root sequents for checking against the class of sound rules that satisfies

$$gleft_F \triangleq map\langle f \rangle$$

$$where\ f(a, b) = \pi_2$$

and the definition of the class is

$$prfsEtcl_C = \mathcal{I}_2(eprfl_C, prflg_C \cup prfg_C)$$

The above definition is for (among other things) the class of provable sequents, but this is not quite enough to be useful. As it stands so far, what has been defined is a completely bottom up system; to prove a theorem it is necessary to have all the subproofs constructed beforehand. One possibility would be to do what is described in the presentation of SP, and prove a suitable lemma for each rule. That approach is not so suitable here, since the set of rules has not been completely defined yet, and anyway, there are a great many more of them. However, a single lemma can do a lot of the preliminary work:

$$\exists pl\ [(pl, \text{'prfl'}) \in prfsEtcl_C \wedge (gleft_F\ of\ pl, thm) \in rules_C] \rightarrow$$

$$\exists p\ [(p, \text{'prf'}) \in prfsEtcl_C \wedge \pi_2\ of\ p = thm]\,,$$

which can be paraphrased in English as 'If there exist proofs of the subgoals of an instance of a rule, then there exists a proof of the goal of the rule'. This provides a facility for top down reasoning.

§7 Logical rules

With a general framework for a sequent calculus system sketched out it is possible to define the rules for a particular theory, instead of talking about them in the abstract, and the following section discusses how to do that.

One of the interesting things about this section is the slightly non-standard form (and presentation) of the rules; collections of hypotheses are represented as lists and a principle formula to the left of the sequent has to be at the head of that list, but the usual structural rules for exchange, weakening and contraction are merged in to the other rules (in particular the left weakening rule is sufficient by itself, as it is described here) in a way that more resembles presentations that use sets to hold the hypothesis collection (e.g., for two lists a, b, $(a, b) \in \mathit{subset}_C$ if every member of a is a member of b).

7.1 Propositional rules

Two rules that are ubiquitous in sequent calculus presentations are the cut and basic rule schemas, so I will use them as paradigms for the rule definitions. The basic rule schema is:

$$\frac{}{\Gamma \vdash \boldsymbol{A}}\ \mathsf{basic} \qquad \text{where} \qquad \boldsymbol{A} \in \Gamma$$

while the cut rule schema is

$$\frac{\Gamma \vdash \boldsymbol{A} \qquad \Delta \vdash \boldsymbol{B}}{\mathrm{E} \vdash \boldsymbol{B}}\ \mathsf{cut} \qquad \text{where} \qquad \begin{array}{l} \Gamma \subset \mathrm{E} \\ \Delta \subset \mathrm{E} \cup \{\boldsymbol{A}\} \end{array}$$

This slightly unusual formulation is another small intrusion of pragmatism: it is just easier to add a formula to E than to remove it from Δ. The concrete formulations are:

$$\begin{array}{l} \mathit{basic}_C \triangleq \mathsf{tup}\ (O, (a, b, \text{'seq'})) \\ \qquad \mathsf{where}\ (b, a) \in \mathit{member}_C \\ \qquad\qquad (O, (a, b, \text{'seq'})) \in \mathit{ruleb}_C \end{array}$$

and

$$\begin{array}{l} \mathit{cut}_C \triangleq \mathsf{tup}\ (O, (g, a, \text{'seq'}), (d, b, \text{'seq'}), (e, b, \text{'seq'})) \\ \qquad \mathsf{where}\ (g, e) \in \mathit{subset}_C \\ \qquad\qquad (d, (e, a)) \in \mathit{subset}_C \end{array}$$

The general scheme for rules that deal with the right-hand side of the sequent is also simple. Consider one of the rules for $\vee$ on the right:

$$\frac{\Gamma \vdash \boldsymbol{A}}{\Delta \vdash \boldsymbol{A} \vee \boldsymbol{B}} \vee\text{-r}_1 \qquad \text{where} \qquad \Gamma \subset \Delta$$

which is defined

$$disjRR_C \triangleq \mathsf{tup}\ (O, (g, a, \text{'seq'}), (d, ((a, b), \text{'}\vee\text{'}), \text{'seq'}))$$
$$\mathsf{where}\ (g, d) \in subset_C$$

The rules that deal with the left-hand side of the sequent are not quite so simple, since it is here that the unusual features mentioned above appear. The formulation of the rule for $\vee$ on the left is:

$$\frac{\Gamma \vdash \boldsymbol{C} \qquad \Delta \vdash \boldsymbol{C}}{\mathrm{E},\ \boldsymbol{A} \vee \boldsymbol{B} \vdash \boldsymbol{C}} \vee\text{-l} \qquad \text{where} \qquad \Gamma \subset \{\boldsymbol{A}\} \cup \mathrm{E}$$
$$\Delta \subset \{\boldsymbol{B}\} \cup \mathrm{E}$$

Here the principal formula $\boldsymbol{A} \vee \boldsymbol{B}$ occurs at the head of the hypothesis list in the goal. It would be possible (and, some might argue, cleaner) to use the definition:

$$\frac{\Gamma \vdash \boldsymbol{C} \qquad \Delta \vdash \boldsymbol{C}}{\mathrm{E} \vdash \boldsymbol{C}} \vee\text{-l} \qquad \text{where} \qquad \exists A\, \exists B$$
$$\boldsymbol{A} \vee \boldsymbol{B} \in \mathrm{E} \wedge$$
$$\Gamma \subset \mathrm{E} \cup \{A\} \wedge$$
$$\Delta \subset \mathrm{E} \cup \{B\}$$

But the existential only makes the definition more complex in practice and there is no real gain.

7.2 Predicate rules

The predicate rules are actually even easier than the propositional rules once the work (described in the section on defining the language) of defining the substitution relations has been done. For instance the rule for existential quantification right is:

$$\frac{\Gamma \vdash \boldsymbol{A}[\boldsymbol{t}/\boldsymbol{v}]}{\Delta \vdash \exists \boldsymbol{v}\ \boldsymbol{A}} \exists\text{-r} \qquad \Gamma \subset \Delta$$

This is implemented as:

$$exiR_C \triangleq \mathsf{tup}\ (O, (g, b, \text{'seq'}), (d, ((v, a), \text{'}\exists\text{'}), \text{'seq'}))$$
$$\mathsf{where}\ (b, a, v) \in subinst_C$$
$$(g, d) \in subset_C$$

The other rules follow the same pattern, and are equally easy.

§8 The definition of the theory

Now it is possible to give the definition of a theory schematic in $\mathcal{L}$. First the basic rule set has to be defined.

$$\begin{aligned} logicalrules_C \triangleq\ & cut_C \cup \\ & disjRR_C \cup \\ & \vdots \\ & exiR_C \end{aligned}$$

And the theory T, with the extra rules R can be defined as

$$T\langle R\rangle \triangleq \mathcal{I}_2(basic_C, logicalrules_C \cup (R \cap ruleb_C))$$

(Notice the side condition that any new rules must also fit the conditions imposed by $ruleb_C$ — this ensures that rules cannot accidentally introduce non wellformed formulae). So, for instance the theory of sorted predicate logic itself is simply

$$AK \triangleq T\langle\{O\}\rangle$$

§9 Summary and conclusions

From the presentation above it is clear that the task of formalising a theory in FS_0 is essentially a job of careful programming and specification. The actual specification here consists of about a thousand lines in all. However a substantial part of that is 'support' and could be factored out since it would be equally useful in very different theories (e.g., the facilities developed for substitution, which make up a large part of the specification, could be reused). Also, the work was the first large experience of using FS_0 and like any such trailblazing if it were done again it would be much improved: some parts of the implementation are frankly badly designed.

Comparing the formalisation here with what would have to be done in a lambda calculus implementation (such as *Isabelle*, or the *LF*), the obvious disadvantage of the presentation here is that substitution has to be defined, instead of being available as a primitive. However, designing a substitution mechanism that could be generalised over different languages that followed the same 'style' of presentation should not be difficult, and once this was done the disadvantage would disappear. The advantage of FS_0, as we have said, is that the presentation is much closer to the natural intuition of what a theory is.

Notes

i. Constants in $\mathcal{L}_t$ are treated as functions of arity zero.

ii. Girard does a thorough job of criticising the $\vee$, $\exists$, $\bot$ fragment of natural deduction in §2 and §10 of [30] — his point is that, for doing meta-theory, the elimination rules for these connectives introduce 'parasitic' formulae. This is essentially a complaint that natural deduction does not have the subterm property that is found in the cut-free sequent calculus

iii. Other hand natural deduction is the 'natural' way to present a logic with an *LF* style framework; all the necessary machinery being available already. For instance implication introduction would be defined simply as something like

$$\Pi_{\vartheta,\psi:o} \cdot (T(\vartheta) \to T(\psi)) \to T(\phi \to \psi)$$

iv. In fact for full generality (c.f. Avron's discussion in [4]) the definition of a sequent would be

$$\Gamma \vdash \Delta \qquad \text{where} \qquad \Gamma, \Delta \in \mathbb{M}(\mathcal{L})$$

i.e., multisets. Or, if even this definition is generalised:

$$\Gamma \vdash \Delta \qquad \text{where} \qquad \Gamma, \Delta \in \mathcal{L}^*$$

i.e., lists.

Meta level reasoning and extension

The last chapter described a general framework for first order logic defined in FS_0. The first part of this chapter extends that work by showing how to prove interesting metatheorems about the declared theory. The particular example chosen is the prenex form theorem for AK, the theory formalised in the last chapter.

The second part of this chapter looks, as a contrast, at another formal theory defined in FS_0. The example is chosen to be as different as possible, both in the sort of theory, and in the way it is encoded. So a version of the lambda calculus ($\boldsymbol{\lambda}$), which uses a 'de Bruijn' style of binding mechanism, instead of the more common 'Church' style is presented; then a version of the theory with the same behaviour, but with a much more efficient implementation of substitution, is built and shown to be equivalent.

§1 Prenex normal form

A good example of a meta level result, since it is fairly practical, and also because of the particular issues that formalising its proof raises, is the prenex normal form theorem [68] for classical predicate logic. Reduction of wffs to various normal forms is one of the preparatory techniques of automated theorem proving, and prenex normal form is an initial stage on the way to several of these, such as skolemisation and disjunctive or conjunctive normal form; it is even an end in itself.

DEFINITION 1. *Prenex normal form.* A wff is in prenex normal form if it is quantifier free, or if it is of the form $\boldsymbol{QvA}$ where $\boldsymbol{A}$ is in prenex normal form.

THEOREM 1. *For every wff* $\boldsymbol{A}$ *there exists a wff* $\boldsymbol{B}$ *such that* $\boldsymbol{B}$ *is in prenex normal form, and* $\boldsymbol{B} \leftrightarrow \boldsymbol{A}$ *is provable (in AK).* The proof here is not for for the full set of connectives, but only a minimal *manageable* set: $\forall, \exists, \vee, \neg$ (defined as the set $wff^{min}{}_C$).

1.1 Overview of the proof

Having stated the theorem, it has to be proven. More, it has to be proven of the representation given in FS_0, and using only the facilities that FS_0 supplies. There are several ways that this can be done, depending on what is wanted from the proof. Here what is wanted is a derived rule for AK so that it is possible to make a single step *PF-normalise* (or *prenex-form normalise*):

$$\frac{\Gamma \vdash \boldsymbol{A}'}{\Gamma \vdash \boldsymbol{A}}\ \textit{PF-normalise}$$

where $\boldsymbol{A}'$ is a normal form of $\boldsymbol{A}$, instead of having to invoke a tactic:

$$\begin{array}{l}\Gamma \vdash \boldsymbol{A}' \\ \quad \vdots\ \texttt{PF-normalise} \\ \Gamma \vdash \boldsymbol{A}\end{array}$$

Even now, whether the intention is to add a tactic or a rule, there is already an informal idea of what the final result will be, i.e., what $\boldsymbol{A}'$ will look like for any given $\boldsymbol{A}$. And no more than this is needed since, at first anyway, what is wanted is an assurance that $\boldsymbol{A}$ can be derived from $\boldsymbol{A}'$. The usual way of getting this assurance is actually to build the proof, but this precisely the work that we want to avoid.

INTERLUDE: *Syntactic sugar.* At this point some syntactic sugar is defined in order to make the presentation clearer. From now on, instead of writing $((a, b), \text{`}\rightarrow\text{'})$, we

will sometimes (adopting the notation defined in the last chapter) write $\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{[\![a]\!] \rightarrow [\![b]\!]}}}}}}}$, where the translation into the actual s-expression form is mechanical; the brackets $[\![\cdot]\!]$ imply that the translation of the term should stop at this point and take the contents as already translated. This allows us to treat terms of FS_0 that are schematic over formulae in the object logic in a clear and intuitive way.

For a new rule, what is needed is a function $f_\sim$ such that $f_\sim(\widetilde{\widetilde{\boldsymbol{A}}}) = \widetilde{\widetilde{\boldsymbol{A}'}}$ where $\boldsymbol{A}'$ agrees with the informal notion of a prenex normal form of $\boldsymbol{A}$, and that also satisfies the formal requirement:

$$\vdash_{FS_0} \forall a (a \in {wff^{min}}_C \leftrightarrow \widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\vdash [\![f_\sim(a)]\!] \rightarrow [\![a]\!]}}}}}}} \in AK$$

It would be possible to define $f_\sim$ directly, and prove all that is required of it at once, but dealing directly with functions is not very satisfactory: the proofs are much more complex than they need to be.

The alternative is to proceed in two stages; first define a relation which describes the effect that is wanted, and then construct a function which satisfies this relation. This has the advantage that a lot of the information about the function is formalised at a more abstract level, in a way that will make proofs of other properties much easier.

Following this approach, the proof will be presented in two main sections. First a class that relates wffs to equivalent prenex normal forms in a way that suggests how to construct $f_\sim$ will be constructed, along with a proof that wffs related this way are logically equivalent. Secondly, a function will be constructed, using the structure of the relation as a guide, and we will show that it satisfies the relation.

1.2 The prenex normal form relation

The first thing to do is describe the prenex normal form relation, which is written as $\sim$, and defined by recursive enumeration using the $\mathcal{I}_2(\cdot,\cdot)$ construct of FS_0. A relation is needed such that if a and b are in ${wff^{min}}_C$ and $a \sim b$ then $\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\vdash [\![a]\!] \leftrightarrow [\![b]\!]}}}}}}} \in AK$.

For the sake of legibility in the definition that follows $a \sim b$ will be written instead of $(a, b) \in {\sim_C}$ — the way the relation is actually represented in the class.

The enumeration is based on the structure of the wffs of the minimal subset, so the following should be read as an analysis by cases of that set. It should also be born in mind that the eventual idea is to prove a theorem saying that if $\widetilde{\widetilde{\boldsymbol{A}}} \sim \widetilde{\widetilde{\boldsymbol{B}}}$ then

$\boldsymbol{A} \leftrightarrow \boldsymbol{B}$, so a note will be provided after each rule, if necessary to say what has to be proven.

The simplest case is for atomic propositions, which since they are quantifier free, are already in prenex normal form. So this can be given as a base case of the relation:

$$\overset{b1}{\sim} \triangleq \mathsf{tup}\ a \sim a \quad \mathsf{where}\ a \in \mathit{apred}_C$$

Now each of the possible outermost connectives has to be considered in turn, each generating one or more step cases.

The first, and simplest, case is when the outermost connective is a quantifier. Here just reducing the interior to prenex normal form will be enough, leaving the outermost quantifier as it is. This gives a single rule:

$$\overset{Q}{\sim} \triangleq \frac{a \sim b \quad -}{(a, v, Q) \sim (b, v, Q')} \quad \mathsf{where}\ Q = Q',\ Q \in \mathit{predcon}_C,\ v \in \mathit{var}_C$$

The second possible connective is negation, which divides again into the four possible cases for the interior. If the interior is an atomic predicate then again, since the expression is quantifier free, it is already in prenex normal form:

$$\overset{b2}{\sim} \triangleq \mathsf{tup}\ (a, (O, \text{`}\bot\text{'}), \text{`}\rightarrow\text{'}) \sim (a, (O, \text{`}\bot\text{'}), \text{`}\rightarrow\text{'}) \quad \mathsf{where}\ a \in \mathit{apred}_C$$

If the interior again is a negation then the two cancel out and the prenex normal form of what is left can be taken instead (this is not actually necessary, but it prevents large numbers of negations accumulating):

$$\overset{\neg\neg}{\sim} \triangleq \frac{a \sim b \quad -}{((a, (O, \text{`}\bot\text{'}), \text{`}\rightarrow\text{'}), (O, \text{`}\bot\text{'}), \text{`}\rightarrow\text{'}) \sim b}$$

The third case is negation of a quantifier; in this case the (logical) identity:

$$\neg \boldsymbol{Q v A} \leftrightarrow \overline{\boldsymbol{Q}} \boldsymbol{v} \neg \boldsymbol{A}$$

can be used to push the negation through the quantifier, then the interior can be replaced with the prenex normal form of its negated self:

$$\overset{\neg Q}{\sim} \triangleq \frac{(a,(O,\text{`}\bot\text{'}),\text{`}\rightarrow\text{'}) \sim b}{((a,v,Q),(O,\text{`}\bot\text{'}),\text{`}\rightarrow\text{'}) \sim (b,v,Q')}$$
$$\textsf{where } Q \in predcon_C$$
$$Q' \in predcon_C$$
$$Q \neq Q'$$
$$v \in var_C$$

The final branch of negation is for a disjunction, which is more complex. First get the prenex normal form of the interior, then get the prenex normal form of the negation of this. This is done simply by pushing the negation through the quantifiers like above in the definition of $\sim^{\neg Q}$. To do this, it is first necessary to define a side class $\sim\sim_C$ which helps describe how a negation can be pushed through the outer quantifiers of a wff. This has the property that if $\widetilde{\boldsymbol{A}} \sim\sim \widetilde{\boldsymbol{B}}$ then $\neg \boldsymbol{A} \leftrightarrow \boldsymbol{B}$, and performs the transformation

$$\widetilde{\boldsymbol{Q}_1 v_1 \ldots \boldsymbol{Q}_n v_n \boldsymbol{A}} \sim\sim \widetilde{\overline{\boldsymbol{Q}}_1 v_1 \ldots \overline{\boldsymbol{Q}}_n v_n \neg \boldsymbol{A}}$$

where $\boldsymbol{A}$ is either a disjunction or a negation — i.e., does not start with a quantifier. The definition is as follows:

$$\sim\sim_b \triangleq \textsf{tup } a \sim\sim (a,(O,\text{`}\bot\text{'}),\text{`}\rightarrow\text{'})$$
$$\textsf{where } \pi_2, a \neq \text{`}\forall\text{'}$$
$$\pi_2, a \neq \text{`}\exists\text{'}$$

$$\sim\sim_s \triangleq \frac{a \sim\sim b}{(a,v,Q) \sim\sim (b,v,Q')}$$
$$\textsf{where } Q \in predcon_C$$
$$Q' \in predcon_C$$
$$Q \neq Q'$$
$$v \in var_C$$

$$\sim\sim_C = \mathcal{I}_2(\sim\sim_b, \sim\sim_s)$$

Then the rule defining negated disjunctions is given as:

$$\overset{\neg \vee}{\sim} \triangleq \frac{((a,b),\text{`}\vee\text{'}) \sim c}{(((a,b),\text{`}\vee\text{'}),(O,\text{`}\bot\text{'}),\text{`}\rightarrow\text{'}) \sim c'}$$
$$\textsf{where } c \sim\sim c'$$

All the above are fairly simple and direct. The problem, and the interest, of the prenex normal form theorem comes from the third case: disjunction. Getting the prenex normal form of the component wffs of a disjunction is easy, but getting from them the prenex normal form of the disjunction itself is not. A side condition, like for $\sim^{\neg\vee}$ above, but more complex, is needed, which defines how the two expressions can be merged.

The relation is written $\approx$ and is between a pair of wffs and a wff, rather than two wffs, so that $(\widetilde{\boldsymbol{A}}, \widetilde{\boldsymbol{B}}) \approx \widetilde{\boldsymbol{C}}$ if, when $\boldsymbol{A}$ and $\boldsymbol{B}$ are in prenex normal form, then $\boldsymbol{C}$ is in prenex normal form and $\boldsymbol{C} \leftrightarrow \boldsymbol{A} \vee \boldsymbol{B}$. The transformation that it performs is

$$(\widetilde{\boldsymbol{Q}_1\boldsymbol{v}_1 \ldots \boldsymbol{Q}_m\boldsymbol{v}_m\boldsymbol{A}}, \widetilde{\boldsymbol{Q}_{m+1}\boldsymbol{v}_{m+1} \ldots \boldsymbol{Q}_n\boldsymbol{v}_n\boldsymbol{B}}) \approx \widetilde{\boldsymbol{Q}_1\boldsymbol{v'}_1 \ldots \boldsymbol{Q}_n\boldsymbol{v'}_n(\boldsymbol{A'} \vee \boldsymbol{B'})}$$

or

$$(\widetilde{\boldsymbol{Q}_1\boldsymbol{v}_1 \ldots \boldsymbol{Q}_m\boldsymbol{v}_m\boldsymbol{A}}, \widetilde{\boldsymbol{Q}_{m+1}\boldsymbol{v}_{m+1} \ldots \boldsymbol{Q}_n\boldsymbol{v}_n\boldsymbol{B}}) \approx \widetilde{\boldsymbol{Q}_1\boldsymbol{v'}_1 \ldots \boldsymbol{Q}_n\boldsymbol{v'}_n(\boldsymbol{B'} \vee \boldsymbol{B})}$$

(where $\boldsymbol{A'}$ and $\boldsymbol{B'}$ are versions of $\boldsymbol{A}$ and $\boldsymbol{B}$, alpha converted so as to avoid variable capture) i.e., we include the possibility that $\boldsymbol{A}$ and $\boldsymbol{B}$ may be switched. It is defined by case analysis: there is one base case and two step cases.

The first is where neither of the wffs has a quantifier as its outermost connective, in which case the disjunction of the two is itself in prenex normal form and can be returned immediately

$$\begin{aligned} \approx_b \triangleq \mathsf{tup}\ (a, b) &\approx ((a, b), \text{`}\vee\text{'}) \\ \mathsf{where}\ a &\in {wff^{min}}_C \\ b &\in {wff^{min}}_C \\ \pi_1, a &\neq \text{`}\exists\text{'} \\ \pi_1, a &\neq \text{`}\forall\text{'} \\ \pi_1, b &\neq \text{`}\exists\text{'} \\ \pi_1, b &\neq \text{`}\forall\text{'} \end{aligned}$$

The second case is where the part of the pair on the left does not have a quantifier as its outermost connective, in which case the pair is commuted.

$$\approx_{com} \triangleq \frac{(b, a) \approx c}{(a, b) \approx c}$$

The third case is where the complexity lies; the outermost quantifier of the left hand component of the pair is moved out so that it encompasses both. The trick

is to ensure that in doing this it does not capture any variables that are free in the right hand component. So, in the definition below, a substitution for the variable is carried out to make sure that this cannot happen—but the specification is written in such a way that it takes makes sure that the intentional information discussed in the section on substitution (where the definition wff_C is given in FS_0) is preserved, if possible.

$$\approx_Q \triangleq \frac{(a', b) \approx c \qquad -}{((a, v, Q), b) \approx (c, v', Q)}$$
$$\textsf{where } ((O, (a, v, Q), b), v', \text{'termlst'}) \in nfi_C$$
$$(a', a, v', v) \in subin_C$$

Then the definition of $\approx$ is simply

$$\approx_C \triangleq \mathcal{I}_2(\approx_b, \approx_{com} \cup \approx_Q)$$

This finally allows definition of the last case of $\sim$, for disjunction: get the prenex normal forms of the two disjuncts, then merge them together carefully, moving the quantifiers out as defined in $\approx$:

$$\overset{\vee}{\sim} \triangleq \frac{a \sim a' \qquad b \sim b'}{((a, b), \text{'}\vee\text{'}) \sim c}$$
$$\textsf{where } (a', b') \approx c$$

With all these definitions in place, finally, the relation $\sim_C$ can be defined in FS_0 as:

$$\sim_C \triangleq \mathcal{I}_2(\overset{b1}{\sim} \cup \overset{b2}{\sim}, \overset{Q}{\sim} \cup \overset{\neg\neg}{\sim} \cup \overset{\neg Q}{\sim} \cup \overset{\neg\vee}{\sim} \cup \overset{\vee}{\sim}))$$

1.3 Verifying the relation

Having defined how wffs are related to their prenex normal forms, it is now necessary to prove that one is the logical equivalent of the other; i.e., (informally) that if $a \sim b$ then $a \leftrightarrow b$, or formally, where the framework is FS_0:

$$\vdash_{FS_0} a \sim b \rightarrow \ulcorner \vdash [\![a]\!] \leftrightarrow [\![b]\!] \urcorner \in AK$$

This can be done by induction on the definition of $\sim$.

INTERLUDE: *More Syntactic Sugar.* Again, in order to make what follows easier to read, some further syntactic sugar is introduced. Instead of writing, for instance,

$\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{[\![a]\!] \leftrightarrow [\![b]\!]}}}}}}} \in AK$, where a and b are free FS_0 variables, we will simply write $\vdash a \leftrightarrow b$; it is important to point out that this is not a ground formula in the language of AK. Moreover we will abbreviate presentations of proofs in the encoded language in the same way; presenting them directly, rather than via manipulations in FS_0.

The same sort of convention will be used for the specifications of proofs, so that, for instance, instead of $f(\widetilde{\widetilde{\widetilde{\widetilde{\boldsymbol{Q}[\![v]\!][\![a]\!]}}}})$ we write just $f(\boldsymbol{Q}va)$, which should be read as $f(((a, v), Q))$, where Q is the encoding of the quantifier $\boldsymbol{Q}$.

The base case

The base case is trivial, at least for the definition of AK used here; since in the base case the two sides of the relation are equal; all that is needed is a proof that:

$$\vdash a \leftrightarrow a$$

and this follows almost immediately from the basic rules in the definition.

The step cases

The step case is not trivial. The FS_0 induction axiom gives two hypotheses,

$$\vdash a \leftrightarrow a' \quad \vdash b \leftrightarrow b'$$

for proving the goal

$$\vdash c \leftrightarrow c'$$

Like this, the hypotheses do not appear to be of much use; however, the relation defining the details of the structure of a, a', b, b', c and c' is a five way branch that, when expanded, gives the information necessary for the proof.

Before breaking up the union into its possible cases though, it makes sense to break the hypotheses into more useful pieces, so that even in the beginning there are four subcases:

$$a \vdash a' \quad a' \vdash a \quad b \vdash b' \quad b' \vdash b$$

The first step case

First, and simplest, is the proof of the $\sim^{Q}$ branch. The definition of this only needs the a hypothesis, and expanding and examining the definition of the branch adds detail to the structure of c and c' in the goal to give:

$$\vdash \boldsymbol{Q}va \leftrightarrow \boldsymbol{Q}va'$$

Four variations need to be proven, but they are all essentially similar, so just one is outlined (right implication where $\boldsymbol{Q}$ is $\forall$):

$$\dfrac{\dfrac{\dfrac{a \vdash a' \checkmark}{\forall v\, a \vdash a'}\,\forall\text{-l}}{\forall v\, a \vdash \forall v\, a'}\,\forall\text{-r}}{\vdash \forall v\, a \rightarrow \forall v\, a'}\,\rightarrow\text{-r}$$

(the $\checkmark$ at the top of the proof simply indicates that this subgoal follows from the induction hypothesis, in this case $\widetilde{[\![a]\!] \vdash [\![a']\!]} \in AK$). The syntactic sugar here is perhaps slightly misleading in that this seems to be a simple proof in the declared logic. However this is not the case: remember that a and variable v are actually unquoted variables in FS_0. This means that some of the side conditions on the proofs cannot be taken care of automatically. For instance in the applications of $\forall$-r and $\forall$-l it is necessary to show that $a'[v/v] = a'$, which in the notation of the system is

$$\vdash_{FS_0} (a', a, v, v) \in subin_C \rightarrow a = a',$$

and

$$\vdash_{FS_0} (\widetilde{\forall [\![v]\!]\, [\![a]\!]}, v, \text{`termvar'}) \in nfi_C$$

The second of these can still be disposed of automatically by the system since the tactic used for object level proofs will immediately encounter the quantifier binding v and stop, without having to speculate on the structure of a. For the first, there is no such conveniently isolating quantifier, and the object level tactic will be inadequate to prove the condition. It is necessary instead to appeal to the fact (stated in the last chapter) that the identity substitution is an identity.

The second step case

The next case that has to be proven is $\sim^{\neg\neg}$, but that is similar to the case that has just been described, and so is left in in favour of $\sim^{\neg Q}$.

The definition of $\sim^{\neg Q}$ is more subtle than for $\sim^{Q}$ in that instead of the structure of the goal simply making use of components of the hypothesis, the relationship goes both ways (this is a problem later, when the function $f_{\sim}$ is defined). Only one, the a hypothesis is needed, and it is refined to show that $a \equiv \neg a''$ while the goal refines to:

$$\vdash \neg \boldsymbol{Q} v a'' \leftrightarrow \overline{\boldsymbol{Q}} v a'$$

Again, just one of the the four parts of the goal is outlined (right implication where $\boldsymbol{Q}$ is $\exists$):

$$\dfrac{\begin{array}{c}\neg a'' \vdash a' \checkmark \\ \vdots \\ \forall v\, \neg a'' \vdash \forall v\, a' \\ \vdots\, lqi \\ \neg \exists v\, a'' \vdash \forall v\, a'\end{array}}{\vdash \neg \exists v\, a'' \rightarrow \forall v\, a'} \rightarrow\text{-r}$$

The upper section of this proof is an instance of the proof schema presented for the proof of the $\sim^{Q}$ branch, the lower part is an invocation of the tactic *lqi* (or *left-quantifier-invert*) which pushes the negation through the existential quantifier in the hypothesis, and again uses the identity of substitution.

The third step case

The third branch of the proof is negation of a disjunction, and is proved:

$$\dfrac{\begin{array}{c}c \vdash a \vee b \checkmark \\ \vdots\, cp \\ \neg(a \vee b) \vdash \neg c \qquad \neg c \vdash c' \ ?\end{array}}{\neg(a \vee b) \vdash c'} \textsf{cut}$$

where the goal marked with a '?' is still unproven, and *cp* is a tactic that reduces a sequent to its contrapositive (if this is possible). In fact it is dependent on the fact that:

$$c \sim\!\sim c' \rightarrow \overbrace{\vdash \neg[\![c]\!] \leftrightarrow [\![c']\!]}^{\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim} \in AK \qquad (L1)$$

The final proof of the relation is not direct, since, like above, it has to take account of a side condition, in this case the $\approx$ relationship; but apart from that it is

easy.

$$\dfrac{\dfrac{\dfrac{b \vdash b'\checkmark}{b \vdash a' \vee b'}\vee\text{-r} \quad \dfrac{a \vdash a'\checkmark}{a \vdash a' \vee b'}\vee\text{-r}}{a \vee b \vdash a' \vee b'}\vee\text{-l} \qquad a' \vee b' \vdash c?}{a \vee b \vdash c}\,\textsf{cut}$$

Here the goal marked with a '?' is still unproven. All that is known about the relationship between a' and b' on one hand, and c on the other, is that:

$$(a', b') \approx c$$

Now, all that is needed is to prove that:

$$\vdash_{FS_0} (a, b) \approx c \rightarrow \ulcorner \vdash [\![a]\!] \vee [\![b]\!] \leftrightarrow [\![c]\!] \urcorner \in AK$$

from which the remaining subgoal quickly follows.

Two of the cases are trivial, the base case is

$$\vdash a \vee b \leftrightarrow a \vee b$$

The first step case is: given the hypothesis

$$\vdash a \leftrightarrow b$$

prove

$$\vdash b \leftrightarrow a$$

The forth step case

The final case is most complex. Given the hypotheses:

$$a[v'/v] \vee b \vdash c \quad c \vdash a[v'/v] \vee b$$

the proof in one direction is (for the case of universal quantification):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a[v'/v] \vdash a[v'/v]}\,\textsf{basic}}{\forall v\, a \vdash a[v'/v]}\forall\text{-l}}{\forall v\, a \vdash a[v'/v] \vee b}\vee\text{-r} \quad \dfrac{\dfrac{}{b \vdash b}\,\textsf{basic}}{b \vdash a[v'/v] \vee b}\vee\text{-r}}{\forall v\, a \vee b \vdash a[v'/v] \vee b}\vee\text{-l} \qquad a[v'/v] \vee b \vdash c\checkmark}{\dfrac{\forall v\, a \vee b \vdash c}{\forall v\, a \vee b \vdash \forall v'\, c}\forall\text{-r}}\,\textsf{cut}$$

There are two places in this proof where quantifier rules are used, but in fact only one of them is a problem. The ∀-I rule does not require anything since the substitution does not need to be evaluated. The ∀-r rule, on the other hand, needs to prove that $c[v'/v'] = c$, which follows from the identity of identity substitution, and

$$\vdash_{FS_0} (\widetilde{\widetilde{\forall v\, a \vee b}}, \widetilde{v'}, \text{'termvar'}) \in nfi_C$$

which follows from the side condition on the rule and the fact (easily demonstrated) that:

$$\vdash_{FS_0} v \in var_C \to v' \in var_C \to a \in wff_C \to b \in wff_C \to$$
$$(\widetilde{\widetilde{[\![a]\!] \vee [\![b]\!]}}, v', \text{'termvar'}) \in nfi_C \to$$
$$(\widetilde{\widetilde{\forall [\![v]\!]\, [\![a]\!] \vee [\![b]\!]}}, v', \text{'termvar'}) \in nfi_C$$

The proof in the other direction raises only the same problems.

1.4 Defining and verifying the function

Having defined the relation, a function that satisfies it has to be produced so that suitable prenex normal forms can actually be generated. The equations that define the function can be read off directly from the definition of the relation $\sim$ as:

$$f_\sim((a, v, \text{'}\forall\text{'})) = (f_\sim(a), v, \text{'}\forall\text{'})$$
$$f_\sim((a, v, \text{'}\exists\text{'})) = (f_\sim(a), v, \text{'}\exists\text{'})$$
$$f_\sim(((a, (O, \text{'}\bot\text{'}), \text{'}\to\text{'}), (O, \text{'}\bot\text{'}), \text{'}\to\text{'})) = f_\sim(a)$$
$$f_\sim(((a, v, \text{'}\forall\text{'}), (O, \text{'}\bot\text{'}), \text{'}\to\text{'})) = (f_\sim((a, (O, \text{'}\bot\text{'}), \text{'}\to\text{'})), v, \text{'}\exists\text{'}) \quad (*)$$
$$f_\sim(((a, v, \text{'}\exists\text{'}), (O, \text{'}\bot\text{'}), \text{'}\to\text{'})) = (f_\sim((a, (O, \text{'}\bot\text{'}), \text{'}\to\text{'})), v, \text{'}\forall\text{'}) \quad (*)$$
$$f_\sim(((a \vee b), (O, \text{'}\bot\text{'}), \text{'}\to\text{'})) = f_{\sim\sim}(f_\sim(((a, b), \text{'}\vee\text{'})))$$
$$f_\sim(((a, b), \text{'}\vee\text{'})) = f_\approx(f_\sim(a), f_\sim(b))$$
$$f_\sim(a) = a \quad \textit{otherwise}$$

Where, at the moment it is enough to say of $f_\approx$ and $f_{\sim\sim}$ that the following hold:

$$\vdash_{FS_0} \forall a\, (a \in wff^{min}{}_C \to (a, f_{\sim\sim}(a)) \in \sim\sim_C) \quad (L2)$$
$$\vdash_{FS_0} \forall a \forall b\, (a \in wff^{min}{}_C \to b \in wff^{min}{}_C \to ((a, b), f_\approx(a, b)) \in \approx_C) \quad (L3).$$

The problem is to define a function that satisfies these equations. The equations above are not immediately suitable, since they are not automatically solvable in

my implementation of FS_0 (the equations marked $(*)$ de not have the necessary subformula property). However a simple rewrite, introducing a secondary function $f_{\sim}^{\neg}$, remedies this:

$$\begin{aligned}
f_{\sim}((a,v,Q)) &= (f_{\sim}(a),v,Q)\\
f_{\sim}((a,(O,'\bot'),'\rightarrow')) &= f_{\sim}^{\neg}(a)\\
f_{\sim}(((a,b),'\vee')) &= f_{\approx}(f_{\sim}(a),f_{\sim}(b))\\
f_{\sim}(a) &= a \quad \textit{otherwise}\\
f_{\sim}^{\neg}((a,v,Q)) &= (f_{\sim}^{\neg}(a),v,\overline{Q})\\
f_{\sim}^{\neg}((a,(O,'\bot'),'\rightarrow')) &= f_{\sim}(a)\\
f_{\sim}^{\neg}(((a,b),'\vee')) &= f_{\sim\sim}(f_{\approx}(f_{\sim}(a),f_{\sim}(b)))\\
f_{\sim}^{\neg}(a) &= (a,(O,'\bot'),'\rightarrow') \quad \textit{otherwise}
\end{aligned}$$

Now all that need be done is prove that:

$$\vdash_{FS_0} v \in {wff^{min}}_C \rightarrow (v, f_{\sim}(v)) \in \sim_C$$

Which is done by induction on the structure of ${wff^{min}}_C$. This follows assuming $(L2)$ and $(L3)$.

Having given necessary properties of $f_{\approx}$ and $f_{\sim\sim}$, they now need to be defined so as to satisfy those properties. First, $f_{\sim\sim}$; the equations read off from the relation definition:

$$\begin{aligned}
f_{\sim\sim}(a,v,Q) &= (f_{\sim\sim}(a),v,\overline{Q})\\
f_{\sim\sim}(a) &= (a,(O,'\bot'),'\rightarrow') \quad \textit{otherwise}
\end{aligned}$$

are directly solvable, and the function they produce is equally simply verified to satisfy $(L1)$ — the proof is similar to the main proof here, of the prenex normal form relation. The other function, $f_{\approx}$, is more interesting; it needs to make use of the function noi_F. One possible candidate can be outlined with the following equations:

$$\begin{aligned}
f_{\approx}((a,v,Q),b) = (&sub_F(noi_F(tsort_F(v),((a,b),'\vee')),v,f_{\approx}(a,b))\\
&noi_F(tsort_F(v),((a,b),'\vee')),\\
&Q)\\
f_{\approx}(a,(b,v,Q)) &= f_{\approx}((b,v,Q),a)\\
f_{\approx}(a,b) &= ((a,b),'\vee') \quad \textit{otherwise}
\end{aligned}$$

Again, these are, as they stand, not solvable with the available machinery. But in this case more than a rewrite is needed; another possibility for the function is:

$$\begin{aligned} f_{\approx}((a,v,Q),b) &= (sub_F(noi_F(tsort_F(v),((a,b),\text{`}\vee\text{'})),v,f_{\approx}(a,b)),\\ &\qquad noi_F(((a,b),\text{`}\vee\text{'})),\\ &\qquad Q)\\ f_{\approx}(a,b) &= f'_{\approx}(a,b) \quad otherwise \end{aligned}$$

where

$$\begin{aligned} f'_{\approx}(a,(b,v,Q)) &= (sub_F(noi_F(tsort_F(v),((a,b),\text{`}\vee\text{'})),v,f_{\approx}(a,b)),\\ &\qquad (noi_F(tsort_F(v),((a,b),\text{`}\vee\text{'}))),\\ &\qquad Q)\\ f'_{\approx}(a,b) &= ((a,b),\text{`}\vee\text{'}) \qquad otherwise \end{aligned}$$

And again this rewritten version is solvable, and the resultant function can be shown to satisfy lemma $(L3)$.

§2 De Bruijn indices for the lambda calculus

Above is presented a description in FS_0 of a language and theory for sorted first order logic. As a contrast, in this section is a presentation of the theory of the lambda calculus, $\boldsymbol{\lambda}$, with de Bruijn indices.

In this section the following conventions hold. $\boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{Z}$ (or decorated variants) are variables over terms in the language Λ; $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ are variables over $\mathbb{N}$.

2.1 The specification of the language Λ, and the theory $\boldsymbol{\lambda}$

The language Λ is very simple, being made up only of variables, abstractions and applications, and is defined as follows.

1. A variable is of the form $\boldsymbol{v_i}$ where $\boldsymbol{i}$ is a natural number. All variables are in Λ.
2. If $\boldsymbol{N}$ is in Λ, then the abstraction $\lambda \boldsymbol{N}$ is in Λ.
3. If $\boldsymbol{M}$ and $\boldsymbol{N}$ are in Λ, then the application of $\boldsymbol{M}$ to $\boldsymbol{N}$, written $(\boldsymbol{M}, \boldsymbol{N})$, is in Λ.

An equivalence relation is defined for objects in this language. Most of the rules are simple, defining equivalence on parts to imply equivalence on the whole and are

defined as follows:

$$M = M \tag{$I.1$}$$

$$M = N \rightarrow N = M \tag{$I.2$}$$

$$M = N \rightarrow N = L \rightarrow M = L \tag{$I.3$}$$

$$M = N \rightarrow (M, Z) = (N, Z) \tag{$I.4$}$$

$$M = N \rightarrow (Z, M) = (Z, N) \tag{$I.5$}$$

$$M = N \rightarrow \lambda M = \lambda N \tag{$I.\xi$}$$

The last of these ($I.\xi$), is also known as the rule of weak extensionality. The other rule is β-conversion, which deals with the way that terms are reduced:

$$(\lambda M, N) = \mathsf{sub}(M, N, 0) \tag{$I.\beta$}$$

where the definitions of of $\mathsf{sub}(\cdot,\cdot,\cdot)$ and $\mathsf{lift}(\cdot,\cdot,\cdot)$ are:

$$\begin{aligned} \mathsf{sub}(v_i, N, j) &= \begin{cases} v_i & \text{if } i \neq j \\ \mathsf{lift}(N, j, 0) & \text{if } i = j \end{cases} \\ \mathsf{sub}(\lambda M, N, j) &= \lambda(\mathsf{sub}(M, N, j+1)) \\ \mathsf{sub}((M, N), Z, j) &= (\mathsf{sub}(M, Z, j), \mathsf{sub}(N, Z, j)) \end{aligned} \tag{$*$}$$

and

$$\begin{aligned} \mathsf{lift}(v_i, j, k) &= \begin{cases} v_i & \text{if } i < k \\ v_{i+j} & \text{if } i \not< k \end{cases} \\ \mathsf{lift}(\lambda M, j, k) &= \lambda(\mathsf{lift}(M, j, k+1)) \\ \mathsf{lift}((M, N), j, k) &= (\mathsf{lift}(M, j, k), \mathsf{lift}(N, j, k)) \end{aligned} \tag{$*$}$$

i.e., the lift function increments the index of all free variables in the substituted term by the number of new scopes that it is inside at the place it is substituted in, so that, for substitution purposes, it remains the same.

2.2 Comparision

The most interesting difference between this and the theory above is the binding mechanism. In the earlier theory, this is defined in the classical way, that preserves the intentional information of the names of bound variables. Here, on the other hand, that information has been thrown away. The argument for doing this is that this makes possible faster implementation of substitution and comparision on machines. This is expoited later to allow the definition of substitution to be modified so that it uses a function instead of an r.e. class.

The other differences are obvious: the syntax is much less intricate, and instead of a consequence relation, all that is needed is equality between objects in the language.

2.3 Describing the language

In this section we will try to use the same variable names in the FS_0 definitions as in the description above, i.e, N, M, N, Z are variables that are intended to vary over terms, and i, j, k are varibles that are supposed to vary over encoded natural numbers.

The definition of the language in FS_0 makes use of the class nat_C defined in the chapter that introduced FS_0, and constant labels for variables, abstraction and applications. The first thing to do is define the distinct constants 'var', 'λ' and 'app' so that the syntax can be encoded as follows:

$$\widetilde{v_i} \equiv (\text{'var'}, \tilde{i})$$

$$\widetilde{\lambda N} \equiv (\text{'}\lambda\text{'}, \widetilde{N})$$

$$\widetilde{(M, N)} \equiv (\text{'app'}, (\widetilde{M}, \widetilde{N}))$$

where $\tilde{i}$ is the encoding of the integer i (using the version of the natural numbers, nat_C, defined earlier), and $\widetilde{M}$ and $\widetilde{N}$ are the encodings of N and M. Then the class of variables is:

$$var_C \triangleq \mathsf{tup}\ (\text{'var'}, x)$$
$$\mathsf{where}\ x \in nat_C$$

and the relations for generating abstractions and applications are:

$$abs_C \triangleq \frac{N}{(\text{'}\lambda\text{'}, N)}$$

$$app_C \triangleq \frac{N \qquad M}{(\text{'app'}, (N, M))}$$

so that the class Λ of terms can be defined in FS_0 as:

$$\mathcal{I}_2(var_C, abs_C \cup app_C)$$

2.4 Describing equality in the theory

In $\boldsymbol{\lambda}$ there is only one type of proposition: the equation. This simplifies the description, since equations can be implemented simply as pairs of objects, so that $\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\widetilde{\boldsymbol{M} = \boldsymbol{N}}}}}}} \equiv (\widetilde{\widetilde{\boldsymbol{M}}}, \widetilde{\widetilde{\boldsymbol{N}}})$. The equivalence rules, $I.1$ to $I.\xi$, are easily dealt with, and so are dealt with first.

The base case of the equivalences is $I.1$

$$I.1_C \triangleq \mathsf{tup}\ (N, N)$$
$$\mathsf{where}\ N \in \Lambda$$

and the other cases are:

$$I.2_C \triangleq \frac{(M, N)}{(N, M)}$$

$$I.3_C \triangleq \frac{(N, M) \qquad (M, Z)}{(N, Z)}$$

$$I.4_C \triangleq \frac{(M, M')}{((\text{'app'}, ((\text{'}\lambda\text{'}, M), N)), (\text{'app'}, ((\text{'}\lambda\text{'}, M'), N)))}$$
$$\mathsf{where}\ N \in \Lambda$$

$$I.5_C \triangleq \frac{(N, N')}{((\text{'app'}, ((\text{'}\lambda\text{'}, M), N)), (\text{'app'}, ((\text{'}\lambda\text{'}, M), N')))}$$
$$\mathsf{where}\ M \in \Lambda$$

$$I.\xi_C \triangleq \frac{(M, N)}{((\text{'}\lambda\text{'}, M), (\text{'}\lambda\text{'}, N))}$$

2.5 Describing reduction as an r.e. relation

The simplest way to describe the $\beta-$**cnv** rule is to define an r.e. class describing the relationship. And this can be done easily by reading off the definition of the class from the definition of **sub** above:

$$\begin{aligned}\beta-\textsf{cnv} \triangleq\ & \textsf{tup}\ ((\text{`app'}, ((\text{`}\lambda\text{'}, M), N)), Z)\\ & \textsf{where}\ M \in \Lambda\\ & \qquad N \in \Lambda\\ & \qquad (M, N, O, Z) \in sub_C\end{aligned}$$

The definition of sub_C is:

$$sub_C = \mathcal{I}_2(basesub1_C \cup basesub2_C, stepsub1_C \cup stepsub2_C)$$

where

$$\begin{aligned}basesub1_C \triangleq\ & \textsf{tup}\ ((\text{`var'}, i), \_, j, (\text{`var'}, i))\\ & \textsf{where}\ i \neq j\\ basesub2_C \triangleq\ & \textsf{tup}\ ((\text{`var'}, i), y, i, x)\\ & \textsf{where}\ (y, i, O, x) \in lift_C\\ stepsub1_C \triangleq\ & \frac{(N, M, (O, i), Z)}{((\text{`}\lambda\text{'}, N), M, i, (\text{`}\lambda\text{'}, Z))}\\ stepsub2_C \triangleq\ & \frac{(N, Z, i, M') \qquad (M, Z, i, M')}{((\text{`app'}, (N, M)), Z, i, (\text{`app'}, (N', M')))}\end{aligned}$$

and the definition of $lift_C$ is

$$lift_C = \mathcal{I}_2(baselift1_C \cup baselift2_C, steplift1_C \cup steplift2_C)$$

where

$$\begin{aligned}baselift1_C \triangleq\ & \textsf{tup}\ ((\text{`var'}, i), j, k, (\text{`var'}, i))\\ & \textsf{where}\ i < j\\ baselift2_C \triangleq\ & \textsf{tup}\ ((\text{`var'}, i), j, k, (\text{`var'}, i + j))\\ & \textsf{where}\ i \not< j\\ steplift1_C \triangleq\ & \frac{(M, j, (O, k), N)}{((\text{`}\lambda\text{'}, M), j, k, (\text{`}\lambda\text{'}, N))}\\ steplift1_C \triangleq\ & \frac{(M, j, k, M') \qquad (N, j, k, N')}{((\text{`}\lambda\text{'}, (M, N)), j, k, (\text{`}\lambda\text{'}, (M', N')))}\end{aligned}$$

(where $i < j$ and $i + j$ are abbreviations for $less_F(i, j) = true$ and $plus_F(i, j)$, which are defined in the obvious way for nat_C).

2.6 A first definition of the class $\boldsymbol{\lambda}$

It is now possible to give a definition of the theory $\boldsymbol{\lambda}$ in FS_0 as:

$$\boldsymbol{\lambda} = \mathcal{I}_2(I.1_C \cup I.\beta_C, (I.2_C \cup I.3_C \cup I.4_C \cup I.5_C \cup I.\xi_C))$$

This definition corresponds almost exactly to the definition that might be given in books on $\boldsymbol{\lambda}$, and is nicely declarative, but for practical use it is awkward: the rule of beta reduction is not a single step operation — the class $I.\beta_C$ is not defined using the decidable subset of the operations of FS_0, so it is necessary to derive each substitution by hand to its conclusion.

§3 A recursive solution to substitution in $\boldsymbol{\lambda}$

The definition of $\boldsymbol{\lambda}$ above is easy to understand, but difficult to work with. The major problem is that beta-reduction is not a one step rule application: each beta reduction in a derivation must itself be proven to be valid, by working through the definition of substitution. A much more satisfactory solution would be to define a theory $\boldsymbol{\lambda}'$, where beta-reduction has been modified so as to be explicitly recursive (so that it can be done automatically), which has the property that

$$\vdash_{FS_0} (N, M) \in \boldsymbol{\lambda} \qquad \text{iff} \qquad \vdash_{FS_0} (N, M) \in \boldsymbol{\lambda}'$$

The way $\boldsymbol{\lambda}'$ is constructed is simply by replacing the class $I.\beta_C$ in the definition with

$$\begin{aligned} I.\beta_C{}' \triangleq \mathsf{tup}\ (((\text{`}\lambda\text{'}, M), N), Z) \\ \mathsf{where}\ M \in \Lambda \\ N \in \Lambda \\ f_\beta, (M, N) = Z \end{aligned}$$

and prove that $\forall x(x \in \boldsymbol{\lambda}' \leftrightarrow x \in \boldsymbol{\lambda})$. This follows from the lemma

$$\vdash_{FS_0} \forall A \forall B \forall C \forall C'(C \subset C' \rightarrow \mathcal{I}_2(A \cup C, B) \subset \mathcal{I}_2(A \cup C', B))$$

and the definition of the class, given that $\forall x(x \in I.\beta_C \leftrightarrow x \in I.\beta_C{}')$. So the problem is reduced to proving this, which in turn reduces to the problem of finding a function sub_F such that

$$\vdash_{FS_0} (N, M, O, Z) \in sub_C \leftrightarrow Z = sub_F(N, M)$$

The rest of this section derives such a function.

3.1 Constructing sub_F

First the right to left implication is discussed (using this to construct a definition of sub_F); i.e., showing that

$$(N, M, O, Z) \in sub_C \rightarrow Z = sub_F(N, M) \qquad S1$$

INTERLUDE: *Useful lemmas.* The following definitions and lemmas describe various properties of operations are used in the following section. They are listed here, but they are probably best referred back to when necessary.

First is the function definition $supersc_F$

$$supersc_F((a, b), i) = \begin{cases} subsc_F(b, i) & \text{if } i < len(b) \\ a & \text{if } i \not< len(b) \end{cases} \qquad D1$$

which is written simply as a superscript for the sake of legibility, i.e, $supersc_F(a, i) \equiv a^i$. Then there is the definition of $subsc_F$,

$$subsc_F((b, a), O) = a$$

$$subsc_F((b, a), (r, O)) = subsc_F(b, r)$$

And again this is abbreviated to a subscript, so that $subsc_F(a, i) \equiv a_i$.

The definitions len_F and *count* are quickly given using 'functionals' as

$$len_F = \text{map}\langle K_O \rangle$$

$$count' = inject\langle f \rangle$$

$$where\ f = \text{map}\langle [I, O] \rangle$$

$$count = C[count, [i, K_O]]$$

Which make the necessary lemmas trivial to prove. These are

$$len_F(\text{map}\langle f \rangle(k, l)) = len_F(l) \qquad D2$$

and

$$len_F(count(len_F(l))) = len_F(l) \qquad D3$$

The following, which describe the behaviour of $subsc_F$, are also useful:

$$i < len_F(l) \rightarrow \text{map}\langle f \rangle(l, k)_i = f(l_i, k) \qquad D4$$

$$i < y \rightarrow count(y)_i = i \qquad D5$$

Given the above properties, a first step in the refinement of the definition of sub_F is to set

$$sub_F(x) = sub_F{}'(x)^O$$

where $sub_F{}'$ satisfies the properties

$$sub_F{}'((\text{'var'}, i), M)^j = \begin{cases} lift_F(M, j) & \text{if } i = j \\ (\text{'var'}, i) & \text{if } i \neq j \end{cases} \qquad p1$$

$$sub_F{}'((\text{'}\lambda\text{'}, M), N)^i = (\text{'}\lambda\text{'}, (sub_F{}'(M, N)^{(O,i)})) \qquad p2$$

$$sub_F{}'((\text{'app'}, (M, N)), Z)^i = (\text{'app'}, cro_F(sub_F{}'(M, Z), sub_F{}'(M, Z))^i) \qquad p3$$

and cro_F amd $lift_F$ satisfy the properties

$$cro_F(x, y)^i = (x^i, y^i) \qquad p4$$

and

$$(y, w, O, y') \in lift_C \rightarrow y' = lift_F(y, w) \qquad p5$$

Then S1 follows from

$$(N, M, j, Z) \in sub_C \rightarrow Z = sub_F{}'(N, M)^j \qquad S2$$

which is provable by induction on the structure of sub_C as follows. For the base case there are two possibilities, corresponding to $basesub1_C$ and $basesub2_C$:

$$i \neq j \rightarrow (\text{'var'}, i) = sub_F{}'((\text{'var'}, i), M)^j$$

which follows from $p1$, and

$$i = j \wedge (M, j, O, M') \in lift_C \rightarrow M' = sub_F{}'((\text{'var'}, i), M)^j$$

which follows since, by $p5$

$$M' = lift_F(j, M)$$

There are also two step cases, corresponding to $stepsub1_C$ and $stepsub2_C$. For the first, only one hypothesis is needed

$$sub_F{}'(N, M)^{(O,i)} = Z$$

and the proof that

$$sub_F{}'((\text{'}\lambda\text{'}, N), M)^i = (\text{'}\lambda\text{'}, Z)$$

is immediate from $p2$. While for the second, both

$$N' = sub_F{}'(N, Z)^i \qquad and \qquad M' = sub_F{}'(M, Z)^i$$

are needed to prove that

$$\begin{aligned} sub_F{}'((\text{'app'}, (N, M)), Z)^i &= (\text{'app'}, cro_F(sub_F{}'(N, Z), sub_F{}'(M, Z))^i) && by\ p3 \\ &= (\text{'app'}, (sub_F{}'(N, Z)^i, sub_F{}'(M, Z)^i)) && by\ p4 \\ &= (\text{'app'}, (N', M')) \end{aligned}$$

The properties that are listed for $sub_F{}'$ are not in a form that is automatically solvable with my system, so the next step of refinement is to provide a definition of $sub_F{}'$ that can be solved. This can be done by refining the definition of $sub_F{}'$ itself to the properties

$$\begin{aligned} f'_\beta((\text{'var'}, i), N) &= ((\text{'var'}, i), list(i, (\text{'var'}, i), lift_F(N, O))) \\ f'_\beta((\text{'}\lambda\text{'}, M), N) &= (tl(f_\lambda(f'_\beta(M, N)))) \\ f'_\beta((\text{'app'}, (M, N)), Z) &= f_\times(f'_\beta(M, Z), f'_\beta(M, Z)) \end{aligned}$$

and the properties $p1$–$p3$ can be verified against this definition, given that the component functions have the properties

$$\begin{aligned} f_\lambda(x)^i &= \lambda(x^i) \\ tl(x)^i &= x^{(O,i)} \end{aligned}$$

and

$$(x, list(x, y, i))^j = \begin{cases} y & if\ i = j \\ x & if\ i \neq j \end{cases}$$

Then, finally, the hd, tl and $list$ functions can be verified against the following definitions, which are automatically solvable:

$$\begin{aligned} hd(x, y) &\triangleq \begin{cases} x & if\ y = O \\ z' & if\ y = (z, z') \end{cases} \\ tl(x, y) &\triangleq \begin{cases} (x, O) & if\ y = O \\ (x, z) & if\ y = (z, z') \end{cases} \\ list(x, y, z) &\triangleq \begin{cases} z & if\ x = O \\ (list(i, y, z), y) & if\ x = (O, i) \end{cases} \\ f_\lambda((x, y)) &\triangleq ((\text{'}\lambda\text{'}, x), f'_\lambda(y)) \\ & \quad where\ f'_\lambda(x) \triangleq \begin{cases} O & if\ x = O \\ (f'_\lambda(y), (\lambda, z)) & if\ x = (y, z) \end{cases} \end{aligned}$$

this leaves only cro_F and $lift_F$ to be dealt with (the development of this is found in the appendices).

The other direction

The proof of the inverse of S1

$$Z = sub_F(N, M) \rightarrow (N, M, O, Z) \in sub_C$$

is then a strightforward matter, since it is logically equivalent to

$$Z \neq sub_F(N, M) \vee (Z = sub_F(N, M) \wedge (N, M, O, Z) \in sub_C)$$

which can be tackled by (course of values) induction.

§4 Conclusion

Two metatheoretic proofs are presented in this chapter. In one sense they are quite similar — they both are theorems about the manipulation of bound variables; but in another they are very different — one is a theorem about a sequent calculus presentation of sorted predicate logic while the other is for an equational presentation of the lambda calculus, one uses a traditional binding mechanism while the other uses an very different binding mechanism proposed by de Bruijn.

We can now evaluate the usability of FS_0 as a theory in which it is possible to do 'practical' meta-theory for a variety of systems. Of the two examples, the prenex normal form theorem clearly requires the less work: it went through without many problems, and in fact a lot of the work was very similar to reasoning at the object level; and even when properly meta-theoretic reasoning was required, it was possible by careful selection of the names of the bound variables, to avoid many problems that could have arisen. (This approach of 'stepping round' the problems with bound variables is not unique to this example either — it is usefully employed in the next part of this thesis.)

On the other hand the development of the substitution algorithm for the lambda calculus required a great deal of careful work in order to build the necessary function. However, a lot of this work was simply because the facilities for defining functions that are available in FS_0 are quite difficult to use, so we would argue that this is not a argument against the practicality of the approach, but an argument for a better function facility in FS_0. In fact the initial formalisation of the theory of the lambda calculus was very simple, and the problem was in showing that the equations defining the substitution function had a primitive recursive solution: given easier ways to do this the development would have been much shorter.

The Idea of Reflection

The chapters above discuss in detail the notion of a framework, and the ways that a particular framework logic (FS_0) can be used to exploit meta-level theorems about a particular theory, so that the work of constructing theorems is made easier. To do this, they describe how to formalise the structure of a theory in a 'framework' $\boldsymbol{MT}$ so that it is possible to reason about that theory, and construct meta-theorems for it. This is done by defining a predicate $Pr(\cdot)$ in $\boldsymbol{MT}$, which has the property that for any formula $\boldsymbol{A}$ in the theory, for which $\widetilde{\boldsymbol{A}}$ is the encoding in the framework, then

$$\vdash_T \boldsymbol{A} \qquad \textit{iff} \qquad \vdash_{\boldsymbol{MT}} Pr(\widetilde{\boldsymbol{A}})$$

One point that is implicit in what has been described is that a theory does not need to be very powerful to allow such a predicate to be defined for an arbitrary theory (after all, the theory ($\boldsymbol{MT}$) used in this thesis for defining theories is FS_0, a conservative extension of PRA). This suggests some interesting possible extensions of the notion of meta-theory by 'closing the circle'. For even quite simple theories then, it is possible to set $T \equiv \boldsymbol{MT}$ and if the encoding mechanism is internalised

(indicated by $\ulcorner\cdot\urcorner$), and a predicate $Pr(\cdot)$ can be defined such that

$$\vdash_T \boldsymbol{A} \qquad \textit{iff} \qquad \vdash_T Pr(\ulcorner \boldsymbol{A} \urcorner).$$

A predicate that has this property will be called a *proof predicate* [72], where the left to right implication is called *completeness*, and the right to left, *soundness* (which correspond to faithfulness and adequacy in a framework).

This chapter outlines some of the theoretical results that are associated with proof predicates and this idea of self reference, or 'reflection', and points out how some of them might be exploited in a PDS, especially one based on a framework theory such as FS_0. These results may have interesting practical implications; but, in the form they appear in this chapter, they are difficult to exploit. However, other work on isolating some of the properties of such proof predicates has been done, and this suggests possible ways that some of the effects of self-reference could be produced without all the work that makes the original form so impractical.

§1 Vocabulary and concepts

The first thing to do is to clear up the vocabulary that is to be used in the future, and introduce some of the new basic concepts.

Facilities that allow an object theory to express certain aspects its metatheory via such a proof-predicate are grouped together under the word *reflection* (since a proof predicate, it can be argued, supplies a facility for a theory to reason 'introspectively' about itself). The idea of reflection has been investigated at length by logicians, and some of the results of that investigation that might be useful in practical theorem proving are discussed in this chapter. While doing this, the second issue discussed is how it is possible to make the facilities of a proof predicate available, without having to go to the (enormous) effort of building the facilities needed to encode the theory inside itself each time.

It is not particularly meaningful to talk about 'meta-' and an 'object-' theories in the context of reflection, but there is an analogous distinction between when the theory is being used to prove theorems *about* itself, and when it is being used simply to prove theorems *in* itself. Some new vocabulary is needed for the purposes of making this distinction. Instead of referring to the 'place' where reasoning takes place, it seems easier to refer to a step that marks the transition from one to the

other, i.e., from outside quotation marks to inside, or from inside to outside. These are known as *reflection up* ($R^{\uparrow}$) and *reflection down*($R_{\downarrow}$) [37]; so a proof predicate is any predicate that satisfies the two reflection rules:

$$\frac{\boldsymbol{A}}{Pr(\ulcorner \boldsymbol{A} \urcorner)}\,(R^{\uparrow}) \qquad \frac{Pr(\ulcorner \boldsymbol{A} \urcorner)}{\boldsymbol{A}}\,(R_{\downarrow})$$

When the word 'reflection' is used here it normally means relating quoted and unquoted formulae in this way[i].

The reflective properties of a predicate satisfying these rules are very weak. The only connection between quoted and unquoted terms is via an admissible rule. It is possible to imagine a stronger form of the above though; one example is the schema

$$Pr(\ulcorner \boldsymbol{A} \urcorner) \rightarrow \boldsymbol{A}$$

(which subsumes the the rule ($R_{\downarrow}$) mentioned above). This asserts the simple proposition (given that the proof predicate corresponds to a true theory) that if a formula is provable, then it is true. Such a connection between the 'inside' and the 'outside' of quotation is referred to here as a *reflection principle*, after the definition used by Feferman[23] who says

> 'By a *reflection principle* we understand a description of a procedure for adding to any set of axioms A certain new axioms whose validity follow from the validity of the axioms A and which formally express, within the language of A, evident consequences of the assumption that all the theorems of A are valid.'

§2 Work on self reference

Results about self reference are normally given for theories that contain PRA, e.g., [68], [47], [72]. This is one of the smallest theories for which a lot of the results can be obtained, since it provides sufficient resources to build the needed quotation mechanisms and diagonalisation lemma[ii]. However, since part of the purpose of this chapter is to consider the properties of such quotation mechanisms apart from particular implementation facilities, the necessary operations are discussed directly, instead of in terms of encoding in a theory like PRA.

For quotation and diagonalisation to be possible, a theory needs some way of treating the syntax of formulae of the language as itself a class of terms in the language, and a substitution mechanism, allowing substitution into quoted formulae.

The notation used earlier for terms in the object language declared in a framework is here adapted to denote quoted terms and formulae of the language. So in any T that supports quotation, for every formula $\boldsymbol{A}$ and term $\boldsymbol{t}$, there are distinct constants in the language of T, written $\ulcorner \boldsymbol{A} \urcorner$ and $\ulcorner \boldsymbol{t} \urcorner$ which are the quoted forms. In order to substitute into a quoted formula it is also necessary to have a function that substitutes the value of terms for variables (a subclass of the terms) in quoted formulae. Thus it is also convenient to have distinguished variables in the language; so if $\boldsymbol{v}$ is a variable of the language of T then there is a constant written $\ulcorner \boldsymbol{v} \urcorner$ which is the quoted form. With this,

$$\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{t})$$

is a function in the theory that takes a quoted formula $\ulcorner \boldsymbol{A} \urcorner$, a quoted variable $\ulcorner \boldsymbol{v} \urcorner$, and a term $\boldsymbol{t}$, and is equal to the quoted formula corresponding to $\ulcorner \boldsymbol{A} \urcorner$, only with the encoding of the normal form of $\boldsymbol{t}$ substituted for $\boldsymbol{v}$ in $\boldsymbol{A}$.

By a ground term being in 'normal form' here we mean that it is in a special subclass of the terms which are provably pairwise distinct. The normal form of a term is the unique term in that subclass to which it is provably equivalent.

Given such quotation and substitution mechanisms, it becomes possible to prove the diagonalisation lemma [47], which says that, for any formula $\boldsymbol{A}$ with one free variable $\boldsymbol{v}$, there exists a formula $\boldsymbol{B}$ for which

$$\vdash_T \boldsymbol{B} \leftrightarrow \boldsymbol{A}[\ulcorner \boldsymbol{B} \urcorner / \boldsymbol{v}]$$

Given a theory supplied with at least this property, the first result that follows from this is the Tarski 'no truth definition' theorem [75]. This states that there is no predicate Tr definable in a consistent theory T, having the property

$$\vdash_T \boldsymbol{A} \leftrightarrow Tr(\ulcorner \boldsymbol{A} \urcorner)$$

(known as the Tarski truth schema) since, by the diagonalisation lemma, if $Tr(\cdot)$ is definable, then there is a formula $\boldsymbol{B}$ in T, such that

$$\begin{aligned} \vdash_T \boldsymbol{B} &\leftrightarrow (Tr(\ulcorner \boldsymbol{B} \urcorner) \rightarrow \bot) \\ &\leftrightarrow Tr(\ulcorner \boldsymbol{B} \urcorner) \end{aligned}$$

i.e., T is inconsistent.

The second result is from Gödel [36], and depends on the further property of T: that it supplies enough facilities so that not only quotation[iii], but a proof predicate is defined for the theory, i.e., a predicate, written $\mathfrak{Bew}(\cdot)$, for which

$$\vdash_T \mathfrak{Bew}(\ulcorner \boldsymbol{B} \urcorner) \qquad \textit{iff} \qquad \vdash_T \boldsymbol{B}.$$

His result says that there is a sentence $\boldsymbol{A}$ in T for which neither $\vdash_T \boldsymbol{A}$ nor $\vdash_T \boldsymbol{A} \rightarrow \bot$ is provable (given that T is consistent). Given a proof predicate satisfying the definition above, the result follows by taking $\boldsymbol{A}$ to be the solution for $\boldsymbol{B}$ in $\vdash_T \boldsymbol{B} \leftrightarrow (\mathfrak{Bew}(\ulcorner \boldsymbol{B} \urcorner) \rightarrow \bot)$. Furthermore, as a corollary, he provided a particular example of such a formula: a statement of the consistency of the theory, i.e.,

$$\mathfrak{Bew}(\ulcorner \bot \urcorner) \rightarrow \bot.$$

These are historic results, but they are more destructive than constructive. Tarski's result means that the simplest way of connecting a theory and its meta-theory (i.e., as an 'if and only if' inside the theory) is not possible. Gödel's result says that while it may be possible to 'connect' the theory to itself at the meta-level, this connection is 'weak'. In fact the second part of Gödel's theorem provides as an example of such an unprovable formula, an instance of the example reflection principle cited above. In fact the second incompleteness theorem shows the unprovability in PA of

$$\mathfrak{Bew}_{PA}(\ulcorner 1 = 0 \urcorner) \rightarrow \bot. \qquad (R_{con_{PA}})$$

That it is not possible to derive a contradiction is the most basic property that a theory is expected to have, so this is an instance of a reflection principle that PA will not support. However, a lot of work has been done on what the extent and implications of Gödel's result, and possible ways to circumvent this incompleteness.

§3 Developments of the work of Tarski and Gödel

The work above discusses what happens when a proof predicate for a theory is embedded in the theory itself: even the simplest reflection principles are not provable. However, this only rules out one approach to reflection. There is at least one way forward.

3.1 The work of Turing

The first person to produce an interesting result in this area was Turing [78], who considered what the effect of repeatedly adding simply a formal statement of consistency to a theory such as PA, to which Gödel's results applied. He considered the sequence of theories PA^c_n, where

$$PA^c_0 = PA$$

$$PA^c_{\alpha'} = PA^c_\alpha[\mathfrak{Bew}_{PA^c_\alpha}(\ulcorner\bot\urcorner) \to \bot]$$

$$PA^c_\kappa = \bigcup_{d \prec \kappa} PA^c_d$$

(where κ is a limit ordinal and $'$ indicates ordinal successor) and proved that every true Π^0_1-formula of arithmetic is provable on a recursive progression of theories based on this principle (in fact going no further than $PA^c_{\omega+1}$).

This is an impressive result, but it is not the result that he wanted. As a proto-computer scientist, he was looking for some form of completeness for Π^0_2-formulae (i.e., the sentences that specify functions), but he did not manage that. In his paper he also conjectured that the schema,

$$\mathfrak{Bew}_T(\ulcorner \boldsymbol{A} \urcorner) \to \boldsymbol{A} \qquad (R^-_T)$$

(known as *local reflection*) was a stronger extension than simply the particular instance where $\boldsymbol{A}$ is $\bot$, but he did not manage to show this either.

3.2 The work of Feferman

Turing's work was enormously extended by Feferman[23], who also looked at more general transfinite extensions, using reflection principles.

His first result was a disproof of Turing's conjecture mentioned at the end of the last section. The second result was much more interesting; he considered a generalisation of local reflection to

$$\forall \boldsymbol{v}(\mathfrak{Bew}_T(\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{v})) \to \boldsymbol{A}(\boldsymbol{v})) \qquad (R^+_T)$$

known as *uniform reflection*. and what happened when it was used to extend a system such as PA transfinitely; that is, when

$$T^+_0 = T$$

$$T^+_{\alpha'} = T^+_\alpha[R^+_{T_\alpha}]$$

$$T^+_\kappa = \bigcup_{\iota \prec \kappa} T^+_\iota$$

(where κ is a limit ordinal). This defines a progression of theories T_d where $d \in \mathcal{O}$, i.e., d is a notation for a (Church-Kleene) constructive ordinal. He showed that every true sentence of arithmetic is provable in T_d for some $d \in \mathcal{O}$ and moreover that it is possible to select a path through $\mathcal{O}$ along which all these theorems are provable. Unfortunately, due to the intensional nature of these progressions, it is possible to have two different representations of the same ordinal producing different sets of theorems, and the proof that some paths have the completeness property is not constructive. However this is a substantial result, and does indicate that the area is worth pursuing[iv].

3.3 The work of Kreisel and Lévy

A different result is available from work done by Kreisel and Lévy. While the work above tells what happens at the limit of various reflection schemas, in [50] they examine what happens even after a single iteration of uniform reflection. For current purposes, the interesting results are quantification of the increase in proof-theoretic strength of PRA and PA that are achieved with a single instance of uniform reflection. (Using the notation of the last subsection) PRA_1^+ is equivalent to PA and PA_1^+ is equivalent to PA with transfinite induction for ϵ_0.

3.4 The work of Gödel

Another possible benefit that such non-conservative extensions might provide is mentioned by Gödel himself, in a short note [35], which is developed further by Ehrenfeucht in [22]. Given an arbitrary recursive function $f(\cdot)$, and functions $l_T(\cdot)$ and $l_{T[A]}(\cdot)$ which return measures of the length of the shortest proof of a provable proposition in T and $T[A]$, and $T[\neg A]$ is undecidable, there exists a formula $\boldsymbol{B}$ such that $\vdash_T \boldsymbol{B}$ where

$$f(l_{T[A]}(\boldsymbol{B})) < l_T(\boldsymbol{B})$$

The same sorts of questions apply to this result of course: since this is an existential theorem, are there any useful formulae for which this 'speed-up' occurs, or does it only apply to special formulae constructed with prior knowledge of the particular $f(\cdot)$, $l_T(\cdot)$ and $l_{T'}(\cdot)$ that are to be used[v]?

3.5 Conclusions

These results are very interesting to a logician, but this thesis is concerned with practical theorem proving, and the question then is, what do these results offer that might be used in practice, and what are the possible problems.

The facilities that reflection might offer are then: a stronger system, which means that results that were unprovable before might now be easily accessible, and more readable (or even shorter) proofs. Further, these effects are available after single instances of reflection, it is not necessary to reflect infinite numbers of times in order to get useful strengthenings.

The problems, however, are equally obvious, and it is not clear that they do not outweigh the possible gains. The difficulty in making use of reflection would seem to be that defining a suitable predicate $\mathfrak{Bew}(\cdot)$ is a technically complex task, and even after it is finished, how could it be checked so that it actually does what it is supposed to do: given a definition of a supposed 'proof predicate' supplied by a user, how could the system check that it really has the properties that define a proof predicate. The next section looks at how it might be possible to circumvent these problems.

§4 Analysing the proof predicate

The last section described some of the results associated with self-referential sentences in theories, but also pointed out that actually constructing a proof predicate was an intricate task; so intricate that it is not clear that it is practically possible, or at least sufficiently possible that the gains that would result are worth the effort.

However, parallel with the work on the effects of reflective extension that is described above, there has been a lot of work on the abstract properties that any proper proof predicate will have to have. There are several reasons for doing this: the first is simply that the abstract behaviour of something so important in logic is interesting for its own sake. The second is more pragmatic: even theoreticians working on paper (such as Hilbert and Bernays, discussed below) encountered exactly the problems with formalising $\mathfrak{Bew}$ that have just been described. For instance some results in the theory of self reference (such as the second incompleteness theorem itself) are practically impossible to construct if it is necessary to work with a complete, messy, definition of the predicate.

4.1 The Hilbert-Bernays derivability conditions

The first abstraction of a proof predicate away from a concrete definition based on recursively enumerable classes of terms, formulae, and proofs, was presented in Hilbert and Bernays' book [43]. The justification for doing this was to present a proper proof of the second of Gödel's incompleteness theorems. This result is usually described as a formalisation of the first, but this is not how Hilbert and Bernays proved it (this would have been an enormous effort). Instead they showed how the second incompleteness theorem could be reduced to the first, given only a short list of properties (derivability conditions) for their predicate (which will be denoted $\mathfrak{B}(\cdot)$) and which they show any Gödel style proof predicate must satisfy. Their list of derivability conditions is

$$\textit{if } \vdash \boldsymbol{A} \rightarrow \boldsymbol{B} \textit{ then } \vdash \mathfrak{B}(\ulcorner \boldsymbol{A} \urcorner) \rightarrow \mathfrak{B}(\ulcorner \boldsymbol{B} \urcorner) \qquad (D_1^{\mathfrak{B}})$$

$$\textit{if } \vdash \mathfrak{B}(\ulcorner \boldsymbol{A}(\boldsymbol{v}) \rightarrow \bot \urcorner) \textit{ then } \vdash \mathfrak{B}(\ulcorner \boldsymbol{A}(\boldsymbol{t}) \rightarrow \bot \urcorner) \qquad (D_2^{\mathfrak{B}})$$

$$\vdash \boldsymbol{f}(\boldsymbol{v}) = 0 \rightarrow \mathfrak{B}(\ulcorner \boldsymbol{f}(\boldsymbol{v}) = 0 \urcorner) \qquad (D_3^{\mathfrak{B}})$$

(where $\boldsymbol{f}$ is a primitive recursive function, and $\boldsymbol{t}$ is a ground term). As Smorynski points out in the introduction of [72], these are effective for their intended purpose — a proof of the second incompleteness theorem — but they are not really satisfactory as an analysis of the properties of the proof predicate itself (in fact he describes $((D_2^{\mathfrak{B}}))$ and $((D_3^{\mathfrak{B}}))$ as 'moderately bizarre').

4.2 The Löb derivability conditions

Another set of derivability conditions (from now on, simply the derivability conditions and defined for the predicate $Pr(\cdot)$) was presented by Löb in [51]. They are simpler, and arguably much more intuitive:

$$\textit{if } \vdash \boldsymbol{A} \textit{ then } \vdash Pr(\ulcorner \boldsymbol{A} \urcorner) \qquad (D_1)$$

$$\vdash Pr(\ulcorner \boldsymbol{A} \urcorner) \wedge Pr(\ulcorner \boldsymbol{A} \rightarrow \boldsymbol{B} \urcorner) \rightarrow Pr(\ulcorner \boldsymbol{B} \urcorner) \qquad (D_2)$$

$$\vdash Pr(\ulcorner \boldsymbol{A} \urcorner) \rightarrow Pr(\ulcorner Pr(\ulcorner \boldsymbol{A} \urcorner) \urcorner) \qquad (D_3)$$

The first of these says simply that the proof predicate is complete; the second that the predicate is closed under modus ponens, and the third is a formalisation of the first[vi].

These properties, plus substitution, are enough to get Löb's theorem, which was given alongside them, and which says

$$\vdash Pr(\ulcorner \boldsymbol{A} \urcorner) \rightarrow \boldsymbol{A} \quad \textit{iff} \quad \vdash \boldsymbol{A} \tag{LT}$$

or, in its formalised (and equivalent) form,

$$\vdash Pr(\ulcorner Pr(\ulcorner \boldsymbol{A} \urcorner) \rightarrow \boldsymbol{A} \urcorner) \leftrightarrow Pr(\ulcorner \boldsymbol{A} \urcorner) \tag{FLT}$$

These derivability conditions are clearly simpler and more intuitive than those proposed by Hilbert and Bernays, and have been analysed in detail, to the extent that it is possible to say that, to quote Smorynski again,

> 'the [Löb] Derivability Conditions and Löb's Theorem (formalised or not) tell the complete story of [the schematic behaviour of] $Pr(\cdot)$.'

The next section discusses what this means and describes some of the results of the analysis.

§5 The analysis of the Löb conditions

The simple propositional nature of the derivability conditions makes an analysis of them using modal logic particularly attractive, and this has been done in [74] and [72]. If the proof predicate is read as the 'necessary' unary connective ($\Box$), in the traditional modal style, then the derivability conditions can be read as the rule of necessitation and the axioms of the modal logic $K4$. If this is extended with the modal translation of (LT) or (FLT), the logic PRL is defined, which has been studied in detail.

5.1 The logic PRL, and its relationship to PRA

If the language is taken simply as the language of propositional logic augmented with a new unary connective $\Box$, the theory is as follows.

- If $\boldsymbol{A}$ is a propositional tautology in the language of PRL, then $\boldsymbol{A}$ is provable in PRL.
- Every instance of the schema $\Box\boldsymbol{A} \rightarrow \Box(\boldsymbol{A} \rightarrow \boldsymbol{B}) \rightarrow \Box\boldsymbol{B}$ (the modal equivalent of (D_2)) is provable in PRL.
- Every instance of the schema $\Box\boldsymbol{A} \rightarrow \Box\Box\boldsymbol{A}$ (the modal equivalent of (D_3)) is provable in PRL.

- Every instance of the schema $\Box(\Box \boldsymbol{A} \rightarrow \boldsymbol{A}) \rightarrow \Box \boldsymbol{A}$ (the modal equivalent of (FLT) — it would, equally, be possible to have (LT) as a rule instead) is provable in PRL.
- PRL is closed under the rules of necessitation (the modal equivalent of (D_1)),

$$\frac{\boldsymbol{A}}{\Box \boldsymbol{A}}$$

and modus ponens,

$$\frac{\boldsymbol{A} \qquad \boldsymbol{A} \rightarrow \boldsymbol{B}}{\boldsymbol{B}}$$

Solovay, in [74], has given a detailed analysis of PRL, which allows, for instance, the equivalent of soundness for $Pr(\cdot)$:

$$\frac{\Box \boldsymbol{A}}{\boldsymbol{A}}$$

to be demonstrated (as an admissible rule). From this it follows that

$$\vdash_{PRL} \boldsymbol{A} \qquad \textit{iff} \qquad \vdash_{PRL} \Box \boldsymbol{A}$$

which is very suggestive of the definition of a proof predicate given above. Of course, a result like this is no more than suggestive, unless a much closer relationship between PRL and PRA can be demonstrated, but this is what Solovay has done. Let $*$ be defined to be any translation of sentences of PRL into PRA, such that

$$\begin{aligned} (\Box \boldsymbol{A})^* &\equiv Pr(\ulcorner \boldsymbol{A}^* \urcorner) \\ (\boldsymbol{A} \circ \boldsymbol{B})^* &\equiv \boldsymbol{A}^* \circ \boldsymbol{B}^* \\ \bot^* &\equiv \bot \\ \boldsymbol{A} \text{ (atomic)} &\equiv \text{a sentence in the language of arithmetic} \end{aligned}$$

(where $Pr(\cdot)$ is some predicate in PRA satisfying the derivability conditions, and (LT), $\circ$ is any binary propositional connective). Then the first Solovay completeness theorem is

> 'For any proposition $\boldsymbol{A}$ in the language of PRL, $\vdash_{PRL} \boldsymbol{A}$ if and only if for every interpretation $*$, $\vdash_{PRA} \boldsymbol{A}^*$'

§6 Practical implications

The work described above has interesting implications for practical proof development systems. I have said that a concrete proof predicate such as $\mathfrak{Bew}(\cdot)$ is impractically messy to implement, even for the benefits that would result. The work of Löb and Solovay though, implies that there might be a way to avoid having to implement the predicate, by appealing just to what is known about its characteristic behaviour; e.g., by using just the derivability conditions, so that it is not necessary to worry about the practical details of definition. Simply extend the logic with quotation, substitution, and then a new predicate for which the defining axioms are (perhaps a slightly modified, or extended version of) the second and third derivability conditions together with a new rule implementing (D_1) for the predicate (Löb's theorem does not have to be added explicitly, since it follows from substitution, quotation and (D_1)–(D_3) [72]).

That this approach works (i.e., produces the appropriate non-conservativity results) does not necessarily follow; for instance the non-conservativity results could depend on the fact that some particular proof predicate has actually been implemented. In fact Simpson [69] has shown that this approach does not work for local reflection. Since the proof is not published, it is reproduced here.

6.1 The conservativity of 'schematic' local reflection

THEOREM 1. *PRA extended with a local reflection schema, where the proof predicate of the schema is defined only using the derivability conditions and Löb's theorem, is conservative w.r.t. the original theory.*

PROOF: First, define $PRA' \equiv PRA + (D_1) + (D_2) + (D_3) + (FLT)$. Then the proof is by showing that any model of PRA' can be expanded to a model of PRA' extended with all instances of $Pr(\ulcorner \boldsymbol{A} \urcorner) \rightarrow \boldsymbol{A}$. The proof is easily modifiable to the case of PA. Let $\mathfrak{M} = (D, 0, s, +, \times)$ be any model of PRA. This can be expanded to a model $\mathfrak{M}' = (D, 0, s, +, \times, Pr)$ by letting

$$[\![Pr]\!]_{\mathfrak{M}'} = \{v \in D \mid v = [\![\ulcorner \boldsymbol{A} \urcorner]\!]_{\mathfrak{M}} \wedge \vdash_{PRA'} \boldsymbol{A}\}$$

(where $\ulcorner \cdot \urcorner$ is some Gödel numbering scheme, and $[\![t]\!]_{\mathfrak{M}}$ is the interpretation of t in the model $\mathfrak{M}$).

First it is necessary to show that $\mathfrak{M}' \models (D_1)$–$(FLT)$; however before this, one fact about PRA' is needed: the soundness of $Pr(\cdot)$; i.e., $\vdash_{PRA'} \boldsymbol{A}$ iff $\vdash_{PRA'} Pr(\ulcorner \boldsymbol{A} \urcorner)$, which was mentioned where I discuss Solovay's work. With this in hand the proof is as follows, proving each of (D_1)–(FLT) in turn.

(D_1) Proving $\mathfrak{M}' \models (D_1)$ amounts to showing that if $\vdash_{PRA'} \boldsymbol{A}$ then $\mathfrak{M}' \models Pr(\ulcorner \boldsymbol{A} \urcorner)$, which follows directly from the definition of $\mathfrak{M}'$.

(D_2) To prove $\mathfrak{M}' \models Pr(\ulcorner \boldsymbol{A} \urcorner) \rightarrow Pr(\ulcorner Pr(\ulcorner \boldsymbol{A} \urcorner) \urcorner)$, assume $\mathfrak{M}' \models Pr(\ulcorner \boldsymbol{A} \urcorner)$. Then, by definition of $\mathfrak{M}'$ it follows that $\vdash_{PRA'} \boldsymbol{A}$, so by (D_1), $\vdash_{PRA'} Pr(\ulcorner \boldsymbol{A} \urcorner)$, and so $\mathfrak{M}' \models Pr(\ulcorner Pr(\ulcorner \boldsymbol{A} \urcorner) \urcorner)$.

(D_3) To prove $\mathfrak{M}' \models Pr(\ulcorner \boldsymbol{A} \urcorner) \rightarrow Pr(\ulcorner \boldsymbol{A} \rightarrow \boldsymbol{B} \urcorner) \rightarrow Pr(\ulcorner \boldsymbol{B} \urcorner)$ assume $\mathfrak{M}' \models Pr(\ulcorner \boldsymbol{A} \urcorner)$ and $\mathfrak{M}' \models Pr(\ulcorner \boldsymbol{A} \rightarrow \boldsymbol{B} \urcorner)$. Then, by definition of $\mathfrak{M}'$, it follows that $\vdash_{PRA'} \boldsymbol{A}$ and $\vdash_{PRA'} \boldsymbol{A} \rightarrow \boldsymbol{B}$, so, by modus ponens, $\vdash_{PRA'} \boldsymbol{B}$ and so $\mathfrak{M}' \models Pr(\ulcorner \boldsymbol{B} \urcorner)$.

(FLT) To prove $\mathfrak{M}' \models Pr(\ulcorner Pr(\ulcorner \boldsymbol{A} \urcorner \rightarrow \boldsymbol{A}) \urcorner) \rightarrow Pr(\ulcorner \boldsymbol{A} \urcorner)$ assume $\mathfrak{M}' \models Pr(\ulcorner Pr(\ulcorner \boldsymbol{A} \urcorner \rightarrow \boldsymbol{A}) \urcorner)$. Then $\vdash_{PRA'} Pr(\ulcorner \boldsymbol{A} \urcorner) \rightarrow \boldsymbol{A}$, and so by (D_1) and (FLT), it follows that $\vdash_{PRA'} Pr(\ulcorner \boldsymbol{A} \urcorner)$ and so, by the soundness of $Pr(\cdot)$, $\vdash_{PRA'} \boldsymbol{A}$ which gives, by the definition, that $\mathfrak{M}' \models Pr(\ulcorner \boldsymbol{A} \urcorner)$.

Since any model of PRA can be expanded to a model of PRA', it is only necessary to show that $\mathfrak{M}' \models Pr(\ulcorner \boldsymbol{A} \urcorner) \rightarrow \boldsymbol{A}$ to complete the proof. Suppose $\mathfrak{M}' \models Pr(\ulcorner \boldsymbol{A} \urcorner)$. Then $\vdash_{PRA} \boldsymbol{A}$ (by definition of $\mathfrak{M}'$) so $\mathfrak{M}' \models \boldsymbol{A}$, as required.

6.2 Uniform reflection

Simpson's proof does not generalise to the case of uniform reflection though, which is anyway a more interesting rule. And it is possible to show non-conservativity, for uniform reflection, at least for simple theories. The next section addresses some of the more practical issues involved in implementing such a schematic reflection rule as an extension.

§7 Implementing an abstract proof predicate

Unfortunately, the discussion above does not treat of some of the practical aspects of implementing such a proof predicate. Two points are particularly important: there is no discussion of how to deal with the problem of what to do in quoted contexts, and the exact form of the rules as they should be found in the system is not given. This section discusses these issues, especially how the first question affects the answer to the second.

7.1 Variables in quoted formulae

Since uniform reflection is to be exploited here, it is necessary to be able to deal with applications of proof predicates to quoted formulae which have *unquoted* components. The antecedent of the uniform reflection schema is of the form:

$$Pr(\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{y} \urcorner, \boldsymbol{v})).$$

In order to reason about this we will want to be able to do things such as instantiate it with arbitrary terms, and it is important to be careful about how this is done. Remember that the $\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{t})$ function substitutes the encoding of the *normal form* of $\boldsymbol{t}$ for $\boldsymbol{v}$ in $\boldsymbol{A}$. So if the meaning of $\boldsymbol{t}$ is some number n, its normal form is $s(\cdots s(0)\cdots)$, so that

$$\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{t}) = \ulcorner \boldsymbol{A}[s(\cdots s(0)\cdots)/\boldsymbol{v}] \urcorner$$

(note that it is not necessarily the case that $\ulcorner \boldsymbol{A}[s(\cdots s(0)\cdots)/\boldsymbol{v}] \urcorner = \ulcorner \boldsymbol{A}[s(\boldsymbol{t})/\boldsymbol{v}] \urcorner$). The question then is what sort of definition should $\mathsf{sub}(\cdot,\cdot,\cdot)$ be given in the theory, since it will have to be used with non-ground terms. The approach that has been discussed so far, of making use of only the necessary properties of a function can again be exploited here to give a set of axioms defining the behaviour. The idea is to allow only that outer part of a term that will not change its structure, no matter what further substitution and normalisation operations are performed on it, to be moved into the quoted term, in a substitution operation; i.e., given a term $s(t)$ for instance, the outer $s(\cdot)$ can be 'stripped off', and substituted into a quoted term; for natural numbers only for outer constructors of the form $s(\cdot)$ can this be done. $t + t'$ is not, as it stands, substitutable, since a substitution into it might make it reducable to 0. The first axiom treats the case where $\boldsymbol{v}$ does not occur free in $\boldsymbol{A}$:

$$\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{t}) = \ulcorner \boldsymbol{A} \urcorner$$

if $\boldsymbol{v}$ does not occur free in A; then the axioms for terms that are in head-normal form can be defined (for natural numbers) as:

$$\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, s(\boldsymbol{t})) = \mathsf{sub}(\ulcorner \boldsymbol{A}[s(\boldsymbol{v})/\boldsymbol{v}] \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{t})$$

$$\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, 0) = \ulcorner \boldsymbol{A}[0/\boldsymbol{v}] \urcorner$$

In the same way, the behaviour when the third argument is a quoted object, can be defined, since these are in normal form, as:

$$\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \ulcorner \boldsymbol{B} \urcorner) = \ulcorner \boldsymbol{A}[\ulcorner \boldsymbol{B} \urcorner / \boldsymbol{v}] \urcorner.$$

However, since there is no sensible way of reducing a term such as $\boldsymbol{v}' + \boldsymbol{t}$ where $\boldsymbol{v}'$ is a variable, using the standard definition of $+$, it is not possible to define the behaviour of

$$\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{v}' + \boldsymbol{t})$$

However some way still has to be found so that it is possible to treat these terms in a derivation. This is dealt with below, in the discussion about the rules.

7.2 Rules and axioms for an implementation

It is not possible to add the derivability conditions directly to the theory as a collection of axioms for two reasons: First, one of them is a rule; and secondly the section above shows that we have to be able to treat instances of the proof predicate that contain free variables. Thus it is necessary to parametrise over possible instantiations of a substitution, so here are described the slightly modified versions of the conditions that actually do this (the important question of the safety of such an operation is addressed later in this section).

The second derivability condition

This is the simplest of the derivability conditions to modify, and shows how the others will follow. It is simply:

$$Pr(\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{v}')) \rightarrow Pr(\mathsf{sub}(\ulcorner \boldsymbol{A} \rightarrow \boldsymbol{B} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{v}')) \rightarrow Pr(\mathsf{sub}(\ulcorner \boldsymbol{B} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{v}'))$$

The third derivability condition

Following this, the third derivability condition has to keep track of a substituted term through two levels of quotation, so the parametrised version of it works as follows:

$$Pr(\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{v}')) \rightarrow Pr(\mathsf{sub}(\ulcorner Pr(\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{v}')) \urcorner, \ulcorner \boldsymbol{v}' \urcorner, \boldsymbol{v}'))$$

(where $\boldsymbol{v}'$ is a variable).

The first derivability condition

This condition is a rule, rather than an axiom schema and the problem is how to get a version of the rule that allows the introduction of instances of the proof predicate parametrised over terms. The form of the rule here is:

$$\frac{\vdash \boldsymbol{A}}{\vdash Pr(\mathsf{sub}(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{t}))}$$

(for any variable v, notice that the hypothesis set is empty for the goal and the subgoal). This form of the rule is perhaps not as pleasing as might be for a sequent calculus presentation, where it would be obviously be preferable to have more 'uniform' treatment of $Pr(\cdot)$, with left and right elimination rules for arbitrary contexts. But at the moment we are not investigating the proof theory of the predicate, and this presentation has the advantage of simplicity (and is taken directly from the standard form of the derivability condition, above).

7.3 The soundness of the method

Notably absent from the discussion above is any detailed discussion of whether the method we have suggested is actually sound or not, since we are interested here in the applications of the operation rather than the theory in this thesis. The question of soundness is crucially important though, and as an informal argument for soundness we point out that it is possible to define a quotation and substitution mechanism, and proof predicate in a definitional extension of PRA in such a way that the axioms and rules we have given follow.

This does not give a guarantee of soundness for theories weaker than PRA however, and we would have to exercise caution in applying the method in such a case.

§8 Summary and conclusions

In this chapter I have examined the idea of self-reference in a theory, then the related notion of reflection principles, and how they can be used to strengthen a theory. In doing this I have outlined two complementary bodies of work: that by Turing, Feferman, Kreisel and Lévy, which shows the way that reflection principles strengthen theories in useful ways, and that of Hilbert and Bernays, Löb and Solovay, which examines how it is possible to abstract the properties of a suitable proof predicate away from particular implementational details. Finally, with this work in mind, I have described how it would be possible to modify these ideas so that they can be implemented. This actually requires slight extensions of the work of Löb and Solovay, such as parametrising the derivability conditions for the proof predicate.

The result of doing this is a theory extended in a way that does not make use of dramatically new concepts, but simply on a confidence that the initial theory is true. However, the question of whether this actually makes new theorems provable or not is still not answered; we have pointed out that the form of reflection that is suggested here is *not* as strong as the form discussed by Turing and Feferman, but hopes to achieve some of the same results without having to resort to that sort of amount of work.

The following chapters discuss the actual implementation of a quotation mechanism, and a proof predicate, and shows how it can be applied to a simple theory in order to get some useful non-conservative results.

Notes

i. The concept of reflection here should not be confused with the reflection principles of set theory. These latter correspond to what are known as strong axioms of strong infinity — i.e., they allow the universe defined by a set theory to be enlarged for a particular purpose so as to be a sufficiently accurate approximation to the informal classical universe of sets. [29].

ii. The most important exception to this generalisation is that Gödel presented his original work in the much more powerful theory PA.

iii. Not quotation as it is described here either (as a syntactic extension of the theory), but rather as a definitional extension.

iv. Feferman also managed to get the result Turing was looking for: if $\boldsymbol{A}$ is a true Π^0_2-formula of arithmetic, then an upper bound for a progression for which $\boldsymbol{A}$ is provable is $\omega^2 + \omega + 1$.

v. Since uniform reflection implements a form of the ω-rule, it could be thought of as effecting a speed up from a proof of infinite length to one of finite length.

vi. It is worth pointing out that provability is not the only predicate that is satisfied by the Löb derivability conditions, a version of the property of being a well formed formula, for instance, will also satisfy them.

Implementing Reflection

The last chapter discussed various theoretical results about reflective extensions of formal theories. This chapter discusses the machinery that is needed so that reflection can be used to extend a formal theory implemented in a framework proof development system to a new one, and what problems arise in doing it.

§1 Extending the definition of $\mathcal{L}$

A project to add self-reference to a theory presupposes that a quotation facility is available, so that the theory can manipulate strings corresponding to its own formulae. But even this requires thought and commitments. To make the issue simpler, I will consider here how to extend the language $\mathcal{L}$ of sorted first order logic defined earlier to a language $\mathcal{L}^+$ with quotation and substitution mechanisms built in (substitution here meaning substitution into quoted strings).

1.1 Extending the language of terms

The way quotation was implemented by Gödel is not satisfactory for use here, if only because it is too complex, and too cumbersome; Gödel was presenting a proof, not a facility (and was constrained to presenting it in arithmetic). What is really needed is a facility that is simple to implement and use, and that is designed to need as little equipment as possible.

Three of the sorts of $\mathcal{L}$ are reserved for use as the sorts of quoted formulae quoted terms and quoted variables. So if these special sorts are denoted 'qform', 'qterm' and 'qvar', then it is easy to extend the definition of $\mathcal{L}$ with the extra clauses in the definition of $\mathcal{L}_t$

- If $\boldsymbol{A}$ is a formula, then $\ulcorner \boldsymbol{A} \urcorner$ is a term the sort of which is the reserved sort of quoted formulae.
- If $\boldsymbol{t}$ is a formula, then $\ulcorner \boldsymbol{t} \urcorner$ is a term the sort of which is the reserved sort of quoted terms.
- If $\boldsymbol{v}$ is a variable, then $\ulcorner \boldsymbol{v} \urcorner$ is a term, the sort of which is the reserved sort of quoted variables.

The question then is: 'how should this be implemented so that the machinery that has already been built is disrupted as little as possible?'

Remember that a variable is represented in the encoding of $\mathcal{L}$ as

$$_{i}\boldsymbol{v}_{\boldsymbol{s}} \equiv (O, ((\boldsymbol{s}, \boldsymbol{i}), \text{'var'}))$$

where $\boldsymbol{s}$ is the sort of the variable, and $\boldsymbol{i}$ is the index, and, importantly, that these are not constrained to be anything in particular; a function application has a similar format. In order to maintain consistency with this format, quoted objects are defined as follows. Given a new constant 'quote', the new terms are defined to have the format:

$$\ulcorner \boldsymbol{A} \urcorner \equiv (O, ((\text{'qform'}, \boldsymbol{S}_1), \text{'quote'}))$$
$$\ulcorner \boldsymbol{t} \urcorner \equiv (O, ((\text{'qterm'}, \boldsymbol{S}_2), \text{'quote'}))$$

and

$$\ulcorner \boldsymbol{v} \urcorner \equiv (O, ((\text{'qvar'}, \boldsymbol{S}_3), \text{'quote'}))$$

(where $\boldsymbol{S}_1, \boldsymbol{S}_2$ and $\boldsymbol{S}_3$ are the encodings of $\boldsymbol{A}, \boldsymbol{t}$ and $\boldsymbol{v}$).

This approach has a double advantage over Gödel's, in that it is, first, very easy to install as an extension of the implementation of the original language $\mathcal{L}$, and secondly (and more importantly) the supporting machinery that has been constructed for $\mathcal{L}$ (such as substitution) can easily be adapted for use with the extension.

Now the problem of how to modify the definition of $\mathcal{L}$ can be dealt with. In the definition above, for theories that do not need self reference, the definitions of the language of terms ($\mathcal{L}_t$) and the language of formulae ($\mathcal{L}$) can be kept separate, since the first does not depend at all on the second. This is no longer true in $\mathcal{L}^+$, the two are defined as a pair, together. This fortunately does not involve dramatic restructuring of the definition. All that is necessary is that the definition of formulae be combined into the definition of terms. Once this has been done, it is easy to extend the definition to deal with quoted terms as well.

Remember that the class of terms, lists of terms and atomic predicates was defined as a unit, and then the atomic predicates were projected out of this. Instead of this, the formulae can be constructed at the same time as everything else and then they can be projected out. This is a simple matter of modifying the rules for proposition and predicate construction as follows.

Remember from the chapter 'Defining a Language' that the definition of $\mathcal{L}$ (written wff_C) is

$$wff_C \triangleq \mathcal{I}_2(rpred_C \cup absurds_C,\ propg_C \cup predg_C)$$

so to get the rules for generating formulae in $\mathcal{L}^+$ the new rules need to be defined

$$refwffbase_C \triangleq \mathsf{tup}\ (a, \text{'formula'})$$
$$\mathsf{where}\ a \in absurd_C$$
$$refwffstep_C \triangleq \frac{(a, \text{'formula'}) \qquad (b, \text{'formula'})}{(c, \text{'formula'})}$$
$$\mathsf{where}\ (c, a, b) \in propg_C \cup predg_C$$

(notice that the base case of atomic predicates has not been defined here; but that is dealt with in a moment).

In the old definition of terms, the atomic predicates are defined at the same time as the classes of lists of terms and terms, as

$$rpredsetc_C \triangleq \mathcal{I}_2(bt_C \cup emptytl_C,\ extl_C \cup termg_C \cup rpredg_C)$$

where the subsets are labeled as 'rpred', 'terml' or 'term'. These labels can be conveniently recycled for use here, except that the rule for generating atomic predicates is changed to

$$rpredg_C \triangleq \frac{(l, \text{'terml'}) \quad -}{(((l,p), \text{'rpred'}), \text{'formula'})}$$
$$\text{where } p \in prId_C$$
$$(l, \pi_{11}, p) \in wt_C$$

Then two entirely new rules need to be defined: a base class of quoted variables and a new step case that turns formulae into quoted formulae

$$quotedvariables_C \triangleq \text{tup } ((O, ((\text{'qvar'}, v), \text{'quote'})), \text{'term'})$$
$$\text{where } v \in var_C$$

$$quotedtermgen_C \triangleq \frac{(x, \text{'formula'}) \quad -}{((O, ((\text{'qform'}, x), \text{'quote'})), \text{'term'})}$$

Then the definition of the class containing formulae, lists of terms, and terms, is

$$refwffsEtc_C \triangleq \mathcal{I}_2(quotedvariables_C \cup refwffbase_C,$$
$$refwffstep_C \cup$$
$$quotedtermgen_C \cup$$
$$extl_C \cup$$
$$termg_C \cup$$
$$rpredg_C)$$

and the class of formulae of $\mathcal{L}^+$ is simply the subset of these marked with the label 'formula' on the right

$$refwff_C \triangleq \text{tup } a$$
$$\text{where } (a, \text{'formula'}) \in refwffsEtc_C$$

1.2 Extending substitution

With the new language defined it is necessary to modify the definition of substitution in $\mathcal{L}$ to take care of this, but this is a trivial matter, since all that is necessary is to add new base cases for quoted objects, and say substitution does not enter that.

$$naisub_F(t, a, (O, ((b,t), \text{'quote'}))) = (O, ((b,t), \text{'quote'}))$$

(of course the various theorems about sort preserving properties of substitution have to be reproven as well). Substitution into quoted formulae is dealt with in the next section.

§2 The substitution functions and reflecting up

The next issue is the facilities that have to be supplied so that the special substitution functions can be defined to work properly with reflection up.

2.1 Defining substitution into quoted terms

A substitution function takes as its arguments a quoted term or formula $\ulcorner \boldsymbol{A} \urcorner$, a quoted variable $\ulcorner \boldsymbol{v} \urcorner$, and a term $\boldsymbol{t}$:

$$sub(\ulcorner \boldsymbol{A} \urcorner, \ulcorner \boldsymbol{v} \urcorner, \boldsymbol{t})$$

However there is not just one such function (or even two: one for quoted terms and one for quoted formulae) but an entire class, indexed over the sort of the variable to be substituted in.

If the identifier for substitution is defined as 'sub', then the general schema of a substitution application is

$${}_{\text{'sub'}}f^{\text{'qform'}}_{\langle \text{'qform'},\text{'qvar'},\boldsymbol{s}\rangle}\langle \boldsymbol{S_1}, \boldsymbol{S_2}, \boldsymbol{S_3}\rangle$$

(where the sort of $\boldsymbol{S_3}$ is $\boldsymbol{s}$) which in future is written, more readably,

$$sub_{\boldsymbol{s}}(\boldsymbol{S_1}, \boldsymbol{S_2}, \boldsymbol{S_3})$$

(and even the subscript $\boldsymbol{s}$ may be dropped for the sake of convenience).

However it is still necessary to define the behaviour for each term independently. Remember from the last chapter that the behaviour was defined in terms of the cases when the quoted variable did not occur free in the quoted formula (when the result was an identity substitution), and normal form of a term of a sort otherwise. In the following we will make use of the notation defined directly for $\mathcal{L}$ — the discussion above and in previous chapters explains how this can be translated into s-expressions of FS_0 so nothing is lost, and the exposition is clearer — and in fact we will progressively introduce further abbreviations on top this; this merely reflects the way an application is developed on a machine, where actual structure of the terms is hidden so as to make the output easier to understand.

Equality statements in the language $\mathcal{L}$ are of the form ${}_{\text{'='}}p^{\langle \boldsymbol{s},\boldsymbol{s}\rangle}\langle t_1, t_2\rangle$ where $\boldsymbol{s}$ is the sort of both $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$ (in the rest of this section this will be abbreviated, for the sake of legibility, to $t_1 = t_2$).

Now part of the behaviour of the substitution function can be defined as simply the encoding of all instances of the basic sequent

$$\Gamma \vdash sub(\ulcorner \boldsymbol{A} \urcorner, \ulcorner {}_i\boldsymbol{v}_s \urcorner, \boldsymbol{t}) = \ulcorner \boldsymbol{A} \urcorner \qquad (SI)$$

(where $\boldsymbol{A}$ is a formula (term) of $\mathcal{L}$, ${}_i\boldsymbol{v}_s$ is a quoted variable of sort $\boldsymbol{s}$, and $\boldsymbol{t}$ is a term of sort $\boldsymbol{s}$, and under the condition that ${}_i\boldsymbol{v}_s$ does not occur free in $\boldsymbol{A}$.

The axioms for substitution of quoted terms are as straightforward, since quoted objects only ever come in normal form (there is no structure like $\succ(x)$, with a free$x$ in the middle) and so the specification of the appropriate instances of the basic sequent is

$$\Gamma \vdash sub(\ulcorner \boldsymbol{A} \urcorner, \ulcorner {}_i\boldsymbol{v}_s \urcorner, \boldsymbol{t}) = \ulcorner \boldsymbol{A}' \urcorner \qquad (SQ)$$

(where $\boldsymbol{A}$ is a formula (term), ${}_i\boldsymbol{v}_s$ is a variable, $\boldsymbol{s}$ is in {'qform', 'qterm', 'qvar'} and $\boldsymbol{t}$ is a quoted object of sort $\boldsymbol{s}$). Here the formula (term) $\boldsymbol{A}'$ is $\boldsymbol{A}$ with $\boldsymbol{t}$ substituted for ${}_i\boldsymbol{v}_s$.

The only other form of substitution is for terms in arithmetic. Here there is one sort of terms 'nat', one constant, 0, and one function $succ(\cdot)$. The latter two can be written in $\mathcal{L}$ as ${}_{(O,Zero)}f^{\langle\rangle}_{\text{`}Nat\text{'}}\langle\rangle$ and ${}_{(O,\text{`}succ\text{'})}f^{\langle\text{`}Nat\text{'}\rangle}_{\text{`}Nat\text{'}}\langle\cdot\rangle$, and are abbreviated from now on to *zero* and $succ(\cdot)$.

Since *zero* is already in a suitable form (no substitution or normalisation will change its structure at all) the definition of substitution for it is the same as above for quoted terms

$$\Gamma \vdash sub(\ulcorner \boldsymbol{A} \urcorner, \ulcorner {}_i\boldsymbol{v}_{\text{`nat'}} \urcorner, zero) = \ulcorner \boldsymbol{A}' \urcorner \qquad (SN_b)$$

where $\boldsymbol{A}$ is a formula (term), and $\boldsymbol{A}'$ is the formula (term) $\boldsymbol{A}$ with *zero* substituted through for ${}_i\boldsymbol{v}_{\text{`nat'}}$.

Then, finally, for terms $succ(\cdot)$, i.e., those that are explicitly the successor of another term:

$$\Gamma \vdash sub(\ulcorner \boldsymbol{A} \urcorner, \ulcorner {}_i\boldsymbol{v}_{\text{`nat'}} \urcorner, succ(\boldsymbol{t})) = sub(\ulcorner \boldsymbol{A}' \urcorner, \ulcorner {}_i\boldsymbol{v}_{\text{`nat'}} \urcorner, \boldsymbol{t}) \qquad (SN_s)$$

where $\boldsymbol{A}$ is a formula (term), $\boldsymbol{t}$ is a term of type 'nat' and $\boldsymbol{A}'$ is $\boldsymbol{A}$ with $succ({}_i\boldsymbol{v}_{\text{`nat'}})$ substituted through for ${}_i\boldsymbol{v}_{\text{`nat'}}$.

The encodings of the definitions above then give the classes of sequents for substitution SI_C, SQ_C, SN_{bC} and SN_{sC}.

2.2 The diagonalisation lemma

Given just this facility, it is possible to prove the diagonalisation lemma. If $\boldsymbol{A}$ is a formula which has only one free variable, and that variable is of sort 'qform', then there exists another formula $\boldsymbol{B}$ for which it is provable that

$$\vdash \boldsymbol{B} \leftrightarrow \boldsymbol{A}(\ulcorner \boldsymbol{B} \urcorner)$$

I will only give a quick sketch of the proof [47] (a meta-level proof formalised in FS_0 is not difficult):

Given a formula $\boldsymbol{A}$, with only one free variable, define the following

$$m \equiv \ulcorner \boldsymbol{A}(sub\text{`}_{\text{qform}}\text{'}(y, \ulcorner y \urcorner, y) \urcorner$$
$$\boldsymbol{B} \equiv \boldsymbol{A}(sub\text{`}_{\text{qform}}\text{'}(m, \ulcorner y \urcorner, m)$$

then

$$\begin{aligned}
\vdash \boldsymbol{B} &\leftrightarrow \boldsymbol{A}(sub\text{`}_{\text{qform}}\text{'}(m, \ulcorner y \urcorner, m) \\
&\leftrightarrow \boldsymbol{A}(sub\text{`}_{\text{qform}}\text{'}(\ulcorner \boldsymbol{A}(sub\text{`}_{\text{qform}}\text{'}(y, \ulcorner y \urcorner, y) \urcorner, \ulcorner y \urcorner, m) \\
&\leftrightarrow \boldsymbol{A}(\ulcorner \boldsymbol{A}(sub\text{`}_{\text{qform}}\text{'}(m, \ulcorner y \urcorner, m) \urcorner) \\
&\leftrightarrow \boldsymbol{A}(\ulcorner \boldsymbol{B} \urcorner)
\end{aligned}$$

Substitution and the derivability conditions (which are discussed next) are then enough between them to prove Löb's theorem for any theory built on these facilities.

2.3 A proof predicate

The next stage of the extension is adding a proof predicate. Defining this is not difficult: simply take a distinguished predicate labeled with a new constant 'prf' which takes one parameter of type 'qform'. This has the form:

$$\text{`}_{prf}\text{'}p^{<\text{`qform'}>}\langle\cdot\rangle$$

which is abbreviated in future to $Pr(\cdot)$. The rules for this are (as pointed out above) slightly generalised from what is expected from the derivability conditions.

The rules

So the rules are actually the encodings of

$$\frac{\vdash \boldsymbol{A}}{\vdash Pr(sub(\ulcorner \boldsymbol{A} \urcorner, \ulcorner {}_i\boldsymbol{v}_s \urcorner, {}_{i'}\boldsymbol{v}_s))} \qquad (D_1)$$

$$\begin{aligned} \Gamma \vdash & Pr(sub(\ulcorner \boldsymbol{A} \urcorner, \ulcorner {}_i\boldsymbol{v}_s \urcorner, {}_{i'}\boldsymbol{v}_s)) \rightarrow \\ & Pr(sub(\ulcorner \boldsymbol{A} \rightarrow \boldsymbol{B} \urcorner, \ulcorner {}_i\boldsymbol{v}_s \urcorner, {}_{i'}\boldsymbol{v}_s)) \\ & Pr(sub(\ulcorner \boldsymbol{B} \urcorner, \ulcorner {}_i\boldsymbol{v}_s \urcorner, {}_{i'}\boldsymbol{v}_s)) \end{aligned} \qquad (D_2)$$

$$\begin{aligned} \Gamma \vdash & Pr(sub(\ulcorner \boldsymbol{A} \urcorner, \ulcorner {}_i\boldsymbol{v}_s \urcorner, {}_{i'}\boldsymbol{v}_s)) \\ & Pr(sub(\ulcorner Pr(sub(\ulcorner \boldsymbol{A} \urcorner, \ulcorner {}_i\boldsymbol{v}_s \urcorner, {}_i\boldsymbol{v}_s)) \urcorner, \ulcorner {}_i\boldsymbol{v}_s \urcorner, {}_{i'}\boldsymbol{v}_s)) \end{aligned} \qquad (D_3)$$

(Where $\boldsymbol{t}$ is a term of sort $\boldsymbol{s}$. Notice how the variable ${}_i\boldsymbol{v}_s$ has been 'recycled').

Conservatively extending the theory

It is now possible to describe the first conservative extension of the theory. Recall that in §8 of the chapter on declaring a language, the basic theory of the language $\mathcal{L}$ was defined as:

$$T\langle R \rangle \triangleq \mathcal{I}_2(basic_C, logicalrules_C \cup (R \cap ruleb_C))$$

The first stage of the definition for the extended form is then

$$T^{+'}\langle R \rangle \triangleq \mathcal{I}_2(basic_C \cup SR_C, logicalrules_C \cup (D1_C \cup R) \cap ruleb_C))$$

where

$$SR_C \triangleq (SI_C \cup SQ_C \cup SN_{bC} \cup SN_{sC} \cup D2_C \cup D3_C) \cap ruleb_C$$

However, given that a particular instance R' is a ruleset for a theory that contains PRA then $T^{+'}\langle R' \rangle$ is a conservative extension of $T\langle R' \rangle$ since it adds only the Löb derivability conditions and substitution. The next stage, which we hope is non-conservative, is to take out the derivability conditions, and add the reflection schema.

Adding reflection

Since it is possible to derive a contradiction using the first derivability condition and the reflection schema together we must now extend $T^{+'}\langle R\rangle$ to a theory which makes the uniform reflection schema available while forbidding application of the derivability condition. The reflection schema itself is of course an axiom, but it is made available as a rule with no premises

$$\frac{}{\Gamma \vdash \forall_i \boldsymbol{v}_s (Pr(sub(\ulcorner \boldsymbol{A} \urcorner, \ulcorner {}_i\boldsymbol{v}_s \urcorner, {}_i\boldsymbol{v}_s)) \rightarrow \boldsymbol{A}({}_i\boldsymbol{v}_s))} \; \mathit{UniRef}_C$$

and the second stage of the reflection can be defined as:

$$T^{+}\langle R\rangle \equiv \mathcal{I}_2(T^{+'}\langle R\rangle, \mathit{logicalrules}_C \cup \mathit{UniRef}_C \cup (R \cap \mathit{ruleb}_C))$$

i.e., with the derivability conditions carefully removed.

§3 Conclusion

This has been a short chapter. It has shown how to extend the definition of a language (specifically the language $\mathcal{L}$) with quotation, substitution facilities, and how a proof predicate can be defined on top of this. The shortness of the chapter itself is witness that this is a straightforward operation.

Using Reflection

The work above describes a mechanism for adding quotation and a Löb proof predicate to a sorted first order theory. This chapter shows this mechanism being used with the uniform reflection principle for the theory PRA of arithmetic. As defined earlier, this is the theory of arithmetic with same axioms and language (i.e., only the functions and predicates: $succ(\cdot)$, $+$, $\times$ and $<$) as Peano arithmetic but with induction restricted to Σ^0_1-formulae only (defined as the class $sigma1_C$). So, if the rules for these are defined in a class $PRArules_C$ then we have essentially two theories, the smaller is a conservative extension of the ordinary theory of PRA

$$PRA_C \triangleq T^{+'}\langle PRArules_C\rangle$$

and the extended form

$$PRA^+{}_C \triangleq T^{+}\langle PRArules_C\rangle$$

with the two connected together by the $D1_C$ rule.

§1 Course of values induction

It is well known that course of values induction is provable in primitive recursive arithmetic (there is a proof of a similar theorem for FS_0 itself, which is a conservative extension of PRA, in this thesis). But, while an informal argument is not difficult, a formal proof which corresponds to that argument is complicated and fiddly. On the other hand, having extended PRA with a proof predicate and uniform reflection (giving PRA^+), a proof of the same proposition is short and direct.

1.1 Course of values induction

The simplest form of induction is called structural induction, and this is the sort that is usually supplied as primitive in a theory. For instance, in PRA the induction rule is:

$$\frac{\Gamma \vdash P[zero/x] \qquad \Delta, P \vdash P[succ(x)/x]}{\Gamma, \Delta \vdash P}$$

(where P is a Σ_1^0-formula); i.e., given that a property holds for the base components of a sort and that, if it holds for the components of a object of a sort it holds for the object itself, then it holds for all objects of the sort. This is though, is often not the most directly useful induction rule to have. The induction rule that is more often appealed to is what is known as 'course of values' induction, which instead of considering the components of an object, considers the order defined as the transitive closure of the structural relation. A form of this which is admissible in PRA is the axiom schema

$$\forall x(\forall y(y < x \rightarrow A[y/x]) \rightarrow A) \rightarrow \forall x A$$

(where A is a Σ_1^0-formula in which y does not occur free). This schema is well known to be admissible in PRA, but, while the proof is not tricky, it requires a lot of machinery (the course of values theorem proven earlier shows the sort of list processing that is needed in order to build this proof, but they are not available immediately in PRA instead having to be built using the primitives supplied by basic arithmetic). So a formal proof that it is the case would be a demanding task. The next section presents a formal proof of a course of values schema which uses the extensions suggested here to avoid this machinery and provide a simple meta-theoretic (in the sense of earlier chapters) proof in the extended system.

1.2 Formalising the schema

Course of values induction is a schema, not a theorem, so what is needed is a meta-level proof that it is sound, rather than a proof of a particular theorem at the object level. The particular statement then is the theorem

$$\begin{aligned}&\forall a(a \in sigma1_C \rightarrow \forall b(b \in sigma1_C \rightarrow \forall x(x \in var_C \rightarrow \forall y(y \in var_C \rightarrow \\ &\quad (y,a) \in nfi_C \rightarrow (y,a) \in nbi_C \rightarrow (x,a) \in nbi_C \rightarrow \\ &\quad\quad x \neq y \rightarrow (b,a,y,x) \in subin_C \rightarrow \\ &\quad\quad\quad \ulcorner \vdash \forall [\![x]\!](\forall [\![y]\!]([\![y]\!] < [\![x]\!] \rightarrow [\![b]\!]) \rightarrow [\![a]\!]) \rightarrow \forall [\![x]\!][\![a]\!]\urcorner \in PRA^+))))\end{aligned} \tag{Cov}$$

The class nbi_C is the class defining the relation between a term and a formula where the term does not occur bound in the formula — it is defined:

$$\begin{aligned} nbi_C \triangleq \mathsf{tup}\ &(v,a) \\ \mathsf{where}\ &a = \alpha\text{-}cnv_F(v,a) \\ &v \in var_C \\ &a \in wff_C \end{aligned}$$

This statement of the course of values schema is not particularly clear as it stands. So from now on, I will work in the language of the declared theory directly (in the same way as when presenting the prenex normal form theorem in chapter 5). Rephrased in those terms (Cov) is the statement:

$$\vdash_{PRA^+} \forall x(\forall y(y < x \rightarrow b) \rightarrow a) \rightarrow \forall x a \tag{COV}$$

given that a and b are Σ^0_1-formulae, where y is not equal to x and does not occur free in a and $b = a[x/y]$, and neither x nor y occur bound in a.

1.3 Formal proof of course of values induction

Supporting lemmas

The presentation of the proof (as well as the following proofs) makes use of two lemmas, which we describe here.

The first lemma is:

$$\begin{aligned}\vdash_{FS_0} \forall x(x \in var_C \rightarrow \forall y(y \in var_C \rightarrow \\ \forall t(t \in term_C \rightarrow x \neq y \rightarrow \\ \forall a(a \in wff_C \rightarrow naisub_F(y,x,a) = naisub_F(t,x,naisub_F(y,x,a))))))\end{aligned} \tag{IdSub1}$$

I.e., that if a variable y has been (naively) substituted through a formula a for x, then trying to substitute (naively) something through for x subsequently is going to be an identity operation. This follows by induction on a (over the structure of wff_C).

The second lemma is:

$$\begin{aligned}\vdash_{FS_0} \forall x(x \in var_C \rightarrow \forall y(y \in var_C \rightarrow \\ \forall a(a \in wff_C \rightarrow (y,a) \in nfi_C \rightarrow \\ naisub_F(x,y,naisub_F(y,x,a)) = a)))\end{aligned} \tag{IdSub2}$$

which follows in the same way as the last, with induction over the structure of a (though we have to make use of the fact that nfi_C is a decidable class here in order to get the induction).

Now we can proceed with the proof of (COV). This depends on the lemma

$$\vdash_{PRA+} \forall x(\forall x(\forall y(y < x \rightarrow b) \rightarrow a) \rightarrow \forall y(y < x \rightarrow b)) \tag{$L1_{COV}$}$$

as follows, taking care of the same meta-level issues as had to be treated with in the proof of the prenex normal form theorem. First, (COV) is reduced to

$$\forall x(\forall y(y < x \rightarrow b) \rightarrow a) \vdash_{PRA+} \forall x a$$

Then cutting in $(L1_{COV})$ the goal becomes

$$\begin{aligned}&\forall x(\forall x(\forall y(y < x \rightarrow b) \rightarrow a) \rightarrow \forall y(y < x \rightarrow b)) \\ &\forall x(\forall y(y < x \rightarrow b) \rightarrow a) \\ &\quad \vdash_{PRA+} a\end{aligned}$$

Now, observe that the formula

$$\vdash_{PRA+} \forall x(\forall y(y < x \to b) \to a) \to (\forall y(y < x \to b) \to a)$$

results immediately from the first hypothesis. Thus after a couple of proof steps (and keeping the original first hypothesis), the goal reduces to

$$\forall x(\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b))$$
$$\forall x(\forall y(y < x \to b) \to a)$$
$$\vdash_{PRA+} \forall y(y < x \to b)$$

Then by substituting x for x in the first hypothesis we get the situation

$$\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b)$$
$$\forall x(\forall y(y < x \to b) \to a)$$
$$\vdash_{PRA+} \forall y(y < x \to b)$$

and the goal follows easily.

Now what has to be proven is that

$$\vdash_{PRA+} \forall x(\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b))$$

Why this particular (sightly unusual) form of the lemma has been chosen is now clear: it is possible to use $UniRef_C$ to reflect down as follows:

$$\vdash_{PRA+} Pr(sub(\ulcorner \forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b) \urcorner, \ulcorner x \urcorner, x))$$

and then, since the interior is a member of the class $sigma1_C$, apply induction. This gives two subgoals: the base case

$$\vdash_{PRA+} Pr(sub(\ulcorner \forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b) \urcorner, \ulcorner x \urcorner, zero))$$

and the step case

$$Pr(sub(\ulcorner \forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b) \urcorner, \ulcorner x \urcorner, x))$$
$$\vdash_{PRA+} Pr(sub(\ulcorner \forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b) \urcorner, \ulcorner x \urcorner, succ(x)))$$

The base case

This is easy to prove. First, by making use of the rule SN_b, which allows us to show that

$$sub(\ulcorner\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b)\urcorner, \ulcorner x\urcorner, zero)$$
$$= \ulcorner(\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b)[zero/x])\urcorner$$

the goal reduces to

$$\vdash_{PRA^+} Pr(\ulcorner(\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b))[zero/x]\urcorner))$$

Then, making use of the rule $D1_C$ it can be moved into PRA:

$$\vdash_{PRA} (\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b))[zero/x]$$

and by careful reduction of the substitution this reduces to

$$\vdash_{PRA} \forall x(\forall y(y < x \to b) \to a) \to \forall y(y < zero \to (d'))$$

where

$$\vdash_{FS_0} naisub_F(zero, x, b) = d'$$

Then, by eliminating the implication in the goal, and throwing away the resulting hypothesis, the goal becomes

$$\vdash_{PRA} \forall y(y < zero \to d'),$$

And, by removing the universal quantifier by making an identity substitution with y so that it is possible take advantage of the fact that substitution for an identity is an identity operation, and then eliminating the implication, the result follows (without having to worry about the particular structure of d'):

$$\dfrac{\dfrac{\dfrac{}{y < zero \vdash_{PRA} \bot}}{y < zero \vdash_{PRA} d'}}{\vdash_{PRA} y < zero \to d'}$$
$$\vdash_{PRA} \forall y(y < zero \to d'),$$

The step case

The step case is more tricky. At the moment two different terms are being substituted in, and before any progress can be made, this has to be reduced to one. However, by the SN_s rule:

$$sub(\ulcorner\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b)\urcorner, \ulcorner x\urcorner, succ(x))$$
$$= sub(\ulcorner(\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b))[succ(x)/x]\urcorner, \ulcorner x\urcorner, x)$$

and this can be used to replace the goal, so that the two instances of the proof predicate now do have the same term substituted in. Then, using the second derivability condition, this is can be reduced to a requirement to show that

$$\vdash_{PRA+} Pr(sub(\ulcorner(\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b)) \to$$
$$(\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b))[succ(x)/x]\urcorner, \ulcorner x\urcorner, x))$$

Then this reflects up into PRA, using $D1_C$, simply as

$$\vdash_{PRA} (\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b)) \to$$
$$(\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < x \to b))[succ(x)/x]$$

Again, careful reduction of the consequent of the new goal is needed to get it into a useful form:

$$\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < succ(x) \to d)$$

(where $\vdash_{FS_0} d = naisub_F(succ(x), x, b)$) and then making use of *IdSub1* results in

$$\forall x(\forall y(y < x \to b) \to a) \to \forall y(y < succ(x) \to b).$$

Then, by eliminating the implications, the goal is reduced to

$$\forall y(y < x \to b)$$
$$\forall x(\forall y(y < x \to b) \to a)$$
$$\vdash_{PRA} \forall y(y < succ(x) \to b).$$

And, since y does not appear free in the hypothesis list (y does not appear free in a by the conditions in the theorem), it can be used to remove the universal quantifier in the consequence, so the goal reduces again to

$$\forall y(y < x \to b)$$
$$\forall x(\forall y(y < x \to b) \to a)$$
$$\vdash_{PRA} y < succ(x) \to b.$$

Then by eliminating the implication in the goal and then applying the definition of $<$ to the resulting hypothesis, the problem reduces to analysis by cases since, by the definition of $<$, either $y < x$ or $y = x$.

The first case is proven simply by instantiating the first of the hypotheses (and making use of the fact that identity substitution is an identity operation) to

$$y < x \rightarrow b$$

and the result follows easily.

The second case is when $x = y$, and the result obtained by first instantiating the second hypothesis with x to get

$$\forall y(y < x \rightarrow b) \rightarrow a$$

then the antecedent of this can be eliminated against the other hypothesis, reducing the problem to

$$a,\ x = y \vdash_{PRA} b$$

and since $x = y \rightarrow a \rightarrow a[x/y]$, the result follows easily.

1.4 Conclusions

This is a simple example of the sort of advantages a schematic reflexive extension, supported by a capacity for formal meta-theory, can provide. In actual fact, this theorem proves more than this, since the requirement that the formula used in the induction be Σ_1^0 is not actually used — the entire formula, regardless of its structure disappears into a quoted term (or a substitution).

§2 A proof of termination of Ackermann's function

The classic example of a function for which there is no proof of termination in PRA is Ackermann's function. So this is a perfect practical example of the strengthening of PRA that is supplied by PRA^+, and I present here a proof of its termination.

2.1 Ackermann's function

Ackermann's (also sometimes, more accurately, known as Péter's function) function is the standard example given of a function that is not primitive recursive (and therefore, by a theorem of Minc and Parsons [59], not provably computable in PRA). But it is more general than that: the prototype Ackermann function is an interpreter for an enumerable class of recursive functions, and is defined as follows. If the set of functions definable in a theory is enumerable, they can be formed into a list

$$f_0(\cdot), f_1(\cdot), \ldots, f_n(\cdot), f_{n+1}(\cdot), \ldots$$

and, using this list, the (proto) Ackermann's function $Ack_p(\cdot,\cdot)$ can be defined as

$$Ack_p(i,x) = f_i(x)$$

Then, using a simple diagonalisation argument, it is possible to prove that there is no function $f_i(\cdot)$ on the list that corresponds to $Ack_p(\cdot,\cdot)$ (the fact that all the functions on the list are unary, while $Ack_p(\cdot,\cdot)$ is binary, is not important: i.e., there is no function on the list such that

$$f_i(2^x \times 3^y) = Ack_p(x,y)$$

either). This result is usually given for PRA, which is one of the weakest theories for which such a diagonalisation is possible, as an example of a function that is not primitive recursive. But this is not the only theory where the argument works: it can also be applied to the functions that can be defined in HA. It is closely related to the halting problem, the incompleteness theorems, and other results.

The version just described is the 'prototype' version of the function. In most textbooks it is encountered in a much simpler form specific to PRA, where the independence properties are not at all obvious, and it is this form that we will now describe and use.

The first thing to do is define a new predicate in $\mathcal{L}$. This is done simply by defining a new constant name 'ack', using the new predicate $Ack(\cdot_1,\cdot_2,\cdot_3)$ which will be written in future ${}_{`ack'}p^{\langle `int', `int', `int'\rangle}\langle \cdot_1,\cdot_2,\cdot_3\rangle$ Then the Ackermann function can be defined in terms of the three defining conditions

$$\frac{}{\Gamma \vdash_{PRA} Ack(0,y,succ(y))} \; A_1$$

$$\frac{\Delta \vdash_{PRA} Ack(x, succ(zero), z)}{\Gamma \vdash_{PRA} Ack(succ(x), zero, z)} \quad A_2$$

(where $\Delta \subset \Gamma$) and

$$\frac{\Gamma \vdash_{PRA} Ack(succ(x), y, w) \qquad \Delta \vdash_{PRA} Ack(x, w, z)}{\mathrm{E} \vdash_{PRA} Ack(succ(x), succ(y), z)} \quad A_3$$

(where $\Gamma, \Delta \subset \mathrm{E}$). It is possible to show, using a result of Minc and Parsons, that

$$\not\vdash_{PRA} \forall x \forall y \exists z \, Ack(x, y, z)$$

However, in the extended theory PRA^+ we have:

$$\vdash_{PRA+} \forall x \forall y \exists z \, Ack(x, y, z)$$

2.2 The encoding of Ackermann's function

The method of building in PRA the relation that specifies Ackermann's function is technical and not really of interest here — the details can be found in any text on mathematical logic which touches on recursion theory (e.g., [47]). Having decided to leave out the details of the definition, there is the problem of how to make the relationship available anyway. The way that this will be done is simply to use the defining conditions above to extend the theory PRA.

So, if the derivability conditions above are encoded as a set of rules

$$Ack_C \triangleq A_{1C} \cup A_{2C} \cup A_{3C}$$

then we define the two theories

$$PRA[Ack]^{+'} \triangleq T^{+'}\langle PRArules_C \cup Ack_C \rangle$$
$$PRA[Ack]^{+} \triangleq T^{+}\langle PRArules_C \cup Ack_C \rangle$$

Then the formal statement of the goal in FS_0 is:

$$\vdash_{FS_0} \ulcorner \forall_x v_s \forall_y v_s \exists_z v_s \, Ack({}_x v_s, {}_y v_s, {}_z v_s) \urcorner \in PRA[Ack]^{+} \quad A_T$$

(where x, y and z are all distinct). Or, to adopt usual convention of this thesis and write this simply in the object language,

$$\vdash_{PRA[Ack]+} \forall x \forall y \exists z \, Ack(x, y, z) \quad A_T$$

2.3 The proof of Ackermann's function

The problem is then to show that A_T is a theorem. The goal is in a form so that it is possible to make use of $UniRef_C$ immediately, so that the goal becomes

$$\vdash_{PRA[Ack]+} Pr(sub(\ulcorner \forall y \exists z\, Ack(x,y,z) \urcorner, \ulcorner x \urcorner, x))$$

and since the interior of this is quantifier free, it is now possible to apply induction, reducing the problem to the two subgoals; the base case

$$\vdash_{PRA[Ack]+} Pr(sub(\ulcorner \forall y \exists z\, Ack(x,y,z) \urcorner, \ulcorner x \urcorner, zero))$$

and the step case

$$Pr(sub(\ulcorner \forall y \exists z\, Ack(x,y,z) \urcorner, \ulcorner x \urcorner, x)) \vdash_{PRA[Ack]+'} Pr(sub(\ulcorner \forall y \exists z\, Ack(x,y,z) \urcorner, \ulcorner x \urcorner, succ(x))).$$

The base case

The base case is dealt with first. By using the rule SN_b to simplify the substitution, followed an application of $D1_C$, the goal becomes

$$\vdash_{PRA[Ack]+'} \forall y \exists z\, Ack(zero, y, z)$$

and this is proven simply by applying a right universal rule, to get

$$\vdash_{PRA[Ack]+'} \exists z\, Ack(zero, y, z)$$

and then z can be instantiated to $succ(y)$, so that an appeal to the definition of A_1 is all that is needed to finish off.

The step case

The step case is slightly more involved, but again the first step is to simplify substitution in the goal (by removing the $succ(\cdot)$), to get

$$Pr(sub(\ulcorner \forall y \exists z\, Ack(x,y,z) \urcorner, \ulcorner x \urcorner, x)) \vdash_{PRA[Ack]+'} Pr(sub(\ulcorner (\forall y \exists z\, Ack(x,y,z))[succ(x)/x] \urcorner, \ulcorner x \urcorner, x)).$$

then D_2 and $D1_C$ reduces this to

$$\forall y \exists z\, Ack(x,y,z) \vdash_{PRA[Ack]+'} \forall y \exists z\, Ack(succ(x),y,z)$$

Now the goal is of the right form so that induction is allowed on y (it is quantified over a Σ_1-formula to give two subgoals: a base case

$$\forall y \exists z\, Ack(x,y,z) \vdash_{PRA[Ack]+'} \exists z\, Ack(succ(x), zero, z)$$

and a step case

$$\forall y \exists z\, Ack(x,y,z),\ \exists z\, Ack(succ(x),y,z) \vdash_{PRA[Ack]+'} \exists z\, Ack(succ(x),succ(y),z).$$

The base case is proven by instantiating the y in the hypothesis to $succ(zero)$, removing the resulting existential (with the new variable z') and then instantiating the existential in the consequent with z', resulting in

$$Ack(x, succ(zero), z') \vdash_{PRA[Ack]+'} Ack(succ(x), zero, z')$$

which follows by appeal to A_2.

The step case is proven as follows: first the existential hypothesis is eliminated, to get

$$\forall y \exists z\, Ack(x,y,z),\ Ack(succ(x),y,z') \vdash_{PRA[Ack]+'} \exists z\, Ack(succ(x),succ(y),z)$$

Then the y and the z in the other hypothesis are eliminated, the y is instantiated to z', so that the goal is now

$$Ack(x,z',z''),\ Ack(succ(x),y,z') \vdash_{PRA[Ack]+'} \exists z\, Ack(succ(x),succ(y),z).$$

Finally, the existential in the goal is instantiated to z'' and the result follows by appeal to A_3.

§3 PRA^+ contains PA

The two previous described results show that PRA^+ is a considerably stronger theory than PRA. In fact PRA extended with uniform reflection is equal to PA (see, for example, §2.4.17 of [28], and [50]). The proof is quite easy to formalise using the derivability conditions, so it is presented as the final example of proof strengthening in this chapter.

3.1 Coding full induction

A proof that PRA^+ is an extension of PA reduces to a proof that induction over arbitrary formula follows in PRA^+. The only version that is available in PRA is

$$A[0/x] \to \forall x(A \to A[succ(x)/x]) \to \forall x A$$

where A is a Σ_1^0-formula. What the next section shows is that it is possible to delete the side condition on A in the theory PRA^+, which is what is required for PA. The statement of the theorem in FS_0 is as follows:

$$\begin{array}{l} \forall a(a \in wff_C \to \forall b(b \in wff_C \to \forall c(c \in wff_C \to \\ \quad \forall x(x \in var_C \to \\ \qquad (b, a, succ(x), x) \in subin_C \to (c, a, zero, x) \in subin_C \to \\ \qquad\quad \ulcorner \vdash [\![c]\!] \to \forall [\![x]\!]([\![a]\!] \to [\![b]\!]) \to \forall [\![x]\!][\![a]\!]^\urcorner \in PRA^+{}_C)))) \end{array} \qquad (FI)$$

3.2 Proving the result

As usual, I will use the language of the encoded logic directly, since it is much easier to follow the derivation then. The result that is to be shown is

$$\vdash_{PRA^+} c \to \forall x(a \to b) \to \forall x a \qquad (FI)$$

(where $b = a[succ(x)/x]$ and $c = a[zero/x]$). And the proof follows from the lemma

$$\vdash_{PRA^+} \forall x(c \to \forall x(a \to b) \to a) \qquad (L1_{FI})$$

as follows. First the implications in the goal are eliminated, to get:

$$c, \forall x(a \to b) \vdash_{PRA^+} \forall x a. \qquad (FI)$$

Then by cutting in $(L1_{FI})$, instantiating the goal and the lemma with x, and making use of the fact that a identity substitution is an identity operation, the goal is reduced to

$$c, \forall x(a \to b), c \to \forall x(a \to b) \to a \vdash_{PRA+} a$$

Then the result follows by eliminating the implications in the lemma against what is already on the hypothesis list.

This leaves the lemma $(L1_{FI})$ to be proven. Again it is possible to use $UniRef_C$ to reduce the goal to

$$\vdash_{PRA} Pr(sub(\ulcorner c \to \forall x(a \to b) \to a \urcorner, \ulcorner x \urcorner, x)$$

Then this allows Σ_1 induction, which gives the base case

$$\vdash_{PRA+} Pr(sub(\ulcorner c \to \forall x(a \to b) \to a \urcorner, \ulcorner x \urcorner, zero)$$

and the step case

$$Pr(sub(\ulcorner c \to \forall x(a \to b) \to a \urcorner, \ulcorner x \urcorner, x))$$
$$\vdash_{PRA+} Pr(sub(\ulcorner c \to \forall x(a \to b) \to a \urcorner, \ulcorner x \urcorner, succ(x)))$$

The base case

The base case is proven as follows. After using the axiom for substituting *zero* into a string an application of $D1_C$ moves the goal into PRA as:

$$\vdash_{PRA} (c \to \forall x(a \to b) \to a)[zero/x]$$

which by careful reduction (there are no free variables in *zero*, so there is no need for alpha conversion) becomes

$$\vdash_{PRA} (c' \to \forall x(a \to b) \to a')$$

(where $\vdash_{FS_0} naisub_F(zero, x, c) = c'$ and $\vdash_{FS_0} naisub_F(zero, x, a)) = a'$), however we can show that $\vdash_{FS_0} c' = c = a'$ since $\vdash_{FS_0} c = \ulcorner [\![a]\!][zero/x] \urcorner = naisub_F(zero, x, a)$ and by making use of *IdSub2*, reducing the goal to:

$$\vdash_{PRA} c \to \forall x(a \to b) \to c$$

and the result follows almost immediately.

The step case

The step case also proceeds by the use of the rules for substitution into strings. The first thing that needs to be done is to simplify the terms so that they are both talking about substituting the same thing into a string. The rule for substitution of successors, SN_s, which says that

$$sub(\ulcorner c \to \forall x(a \to b) \to a \urcorner, \ulcorner x \urcorner, succ(x))$$
$$= sub(\ulcorner (c \to \forall x(a \to b) \to a)[succ(x)/x] \urcorner, \ulcorner x \urcorner, x)$$

and that, with $D2_C$, and an application of $D1_C$ simplifies the problem to

$$\vdash_{PRA} (c \to \forall x(a \to b) \to a) \to (c \to \forall x(a \to b) \to a)[succ(x)/x].$$

and by careful reduction the consequent of this can be shown to be equal to

$$c' \to \forall x(a \to b) \to b'$$

(where $\vdash_{FS_0} c' = naisub_F(succ(x), x, c)$ and $\vdash_{FS_0} b' = naisub_F(succ(x), x, a)$), and then since it is possible, using *IdSub1*, to show that $\vdash_{FS_0} c = c'$ and $\vdash_{FS_0} b = b'$, can be rewritten to

$$c \to \forall x(a \to b) \to b$$

so that the goal becomes

$$\vdash_{PRA} (c \to \forall x(a \to b) \to a) \to (c \to \forall x(a \to b) \to b).$$

which reduces to

$$a, \forall x(a \to b) \vdash_{PRA} b.$$

Finally, instantiating the hypothesis with x, using the identity of identity substitution and eliminating the implication using the a already on the hypothesis list, the result follows.

§4 The defined proof predicate in *PRA*

In the method above for adding a proof predicate $Pr(\cdot)$ to a theory, though $Pr(\cdot)$ is syntactically an ordinary predicate, it is intended to be equivalent to a Σ_1-formula with one free variable. So we need to be careful about what is a permissible induction. The usual definition of Σ_1 formulae for *PRA* is: arbitrary quantifier free formula (i.e. possibly with negated sub-formulae) prefixed with existential quantifiers (compare the case with FS_0 — a conservative extension of *PRA*— where the induction rule is carefully set up so that induction over quantifier free formulae with negated subformulae is not possible). Now consider the case of a formula with negative occurrence of the proof predicate as a subterm; in the extended theory as defined we would be able to perform an induction, even though in its 'expanded' form the formula would no longer be Σ_1. So there is a possibility that what is supposed to be a conservative extension (i.e., without the reflection schema) would actually be non-conservative.

In the examples here the proof predicate always occurs only positively in formulae to which induction is applied. This does not dispose of the problem of non-conservativity, but we point out that any non-conservativity that results if steps are not taken to prevent induction over negated instances of the predicate, is anyway going to prove only true sentences.

§5 Conclusions

This chapter has presented some small experiments in what is possible with a theory that has been extended with a proof predicate defined using only the derivability conditions. Three results were presented for the theory *PRA* extended using uniform reflection. First a course of values schema was proven, then the definedness of Ackermann's function for all inputs, and finally the extended system was shown to contain *PA*.

The proofs are not difficult to build, and combine use of the proof predicate with use of the meta-level facilities that have been discussed earlier, one facility supporting the other. The examples also show that it is sufficient, at least for a theory such as *PRA* for the proof predicate to be defined in this way, answering the question asked in the last chapter.

Related Work

There has been quite a lot of work done on various aspects of formal meta-theory, its applications, and theorem proving facilities based on reflection. This chapter discusses the relationship of that work to this thesis. Related work can be separated into sections as frameworks, meta-theory, meta-theoretic extension, reflection, and miscellaneous other work.

§1 Frameworks

Frameworks divide into two contrasting traditions. The older of the two, that to which FS_0 belongs, dates back to Post [64]. However it is the newer style, based on ideas from type theory and lambda calculus, that have more often been used as the basis for practical machines, and so it is against this other tradition that, if it is used in practice, FS_0 must be measured. Because of this, in this section I give an overview of work that has been done in type theories, before looking at the Post tradition.

1.1 Type theoretic frameworks

The commonest approach for framework theories that have been suggested for real use, has been various systems of typed lambda calculus that have grown out of early work of Church [15]. Probably the most notable efforts have been *AUTOMATH* and the Edinburgh *LF*.

Automath

The usage of the word 'framework' in work on computer assisted formal reasoning dates back at least to the *AUTOMATH*[12] project at Nijmingen, where it was used to describe the type theories they used to encode formal languages and theories. The ambition of the project was to develop a general system for encoding proofs which they did by reducing the problem of proof checking to the problem of type-checking in a weak type theory, exploiting the facilities of the associated lambda calculus to make the treatment of bound variables easier. In this they were remarkably successful, finally managing, as one example, to formalise a complete textbook on analysis [46]. They actually made use of several theories[20], with various different properties of faithfulness.

The Edinburgh Logical Framework

The Edinburgh *LF* system[38], [39], was inspired by the *AUTOMATH* system only, unlike the *AUTOMATH* languages, it seeks 'to keep a clear distinction between the object- and meta-level, and seeks to handle proof checking for a wider class of systems' §5[39]. The wider class of theories that it can deal with includes Hoare logics, various lambda calculi (including the *LF* itself), and modal logics, all of which have been implemented in it.

One particularly interesting point about the *LF* is that it is a very weak logic (proof theoretically equivalent to the simply typed lambda calculus), and not capable of the sorts of meta-theoretic extensions that are described in this thesis. This weakness is deliberate on the part of the designers; it means that the proof theory is tractable, and so it is possible to develop other sorts of tools for it (such as, for instance, unification [65]).

There are several weaknesses of the *LF* that are not the result of tradeoffs against other advantages somewhere else though. Encoding a modal or temporal logic, for

instance, is not a particularly simple matter; the construction of a correct encoding will require a user to have specialist knowledge of type theory and the proof theory of the logic — and the result will not particularly resemble the normal presentations of those theories. (E.g., the presentation of $S4$ in [5] bears little resemblance to what is found in the standard texts such as [45]). Thus it is not, in general, possible for a typical user to take a presentation and formalise it in the system.

A second weakness is that some theories encountered in practice just do not encode very well. For instance it is not at all clear how one would go about encoding a system such as PRA which has classes of functions with different arities (as it is presented in [47]) so as to be useable.

Nuprl and the Calculus of Constructions

The theories of $AUTOMATH$ and the LF are designed as logics in which it is possible to encode the proofs of a logic effectively. There has also been work on using more powerful type theories such as the Calculus of Constructions (CC) or intuititionistic type theory, which are really designed to be 'foundational', in the same manner. Since LF is a weakening of CC, it is possible to do the same sorts of things in CC. However it is necessary to take care to ensure that the encoding of a logic is faithful, because CC is so powerful.

Recently other work has been done by Basin and Howe in [10] on showing that the particular flavour of ITT used in the *Nuprl* PDS can be used in the same manner as LF.

1.2 Post style systems

As was said earlier, these constitute the earliest tradition of investigations into the idea of an abstract theory for encoding the derivable sentences of a theory, and that to which FS_0 belongs.

The two major presentations are found in Post's work of the first part of this century [64], and Smullyan's thesis [73], which both use the idea of strings. However neither of these systems are very 'practical', since strings, while they are a simple data structure, are not an efficient one to implement. Further, neither of the theories develop the idea of formal meta-theory. The reason for this is that their designers were interested in the theory of the language — it never occurred to them that anyone would be interested actually using a language after it had been declared in the theory.

The other related work in this area does not explicitly discuss what could be called frameworks, but is definitely an important influence: Gödel's results on the incompleteness of theories[36]. This shows that a theory like PA (or PRA) can be used to describe a theory and further, that it is possible to perform substantial meta-theory in such a system. However Gödel's encoding could not, by any stretch of the imagination, be described as practical, even in comparison to Post and Smullyan[i].

PX and the 'Theory of symbolic expressions'

As a footnote to this discussion on 'Post style' frameworks we should mention two other systems than FS_0 that are based on ideas taken from *Lisp*: the *PX* (Program eXtractor) system of Hayashi and Nakano [40], and the *Theory of symbolic expressions* of Sato [82].

PX is a system based on a constructive theory of Feferman called T_0. The theory has the same 'flavour' as FS_0; i.e., it is second order, has similar pairing and projection functions, and a carefully controlled form of comprehension. It is much more powerful though; Sieg proposes it as a system in which to develop parts of intuitionitic analysis. Instead of a second order primitive recursive combinator for instance, it makes the general S and K combinators available, along with comprehension for general first order formulae (with second order parameters). The system is described in detail in [24].

Hayashi and Nakano propose, and implement, a system amalgamating a logic developed from T_0 and a dialect of pure *Lisp*. The result is an untyped theory specially designed for extracting *Lisp* programs from proofs via what they call px-realisability. In their approach they try to separate, as much as possible, the work of verifying termination of functions from other correctness properties; i.e., they introduce a form of Markov's principle via a modal operator working as a double negation connective, that can be eliminated only when it is applied to what they call rank 0 formulae: those they deem to have no computational content.

Their system is better described as a foundational, rather than a framework, theory, since it is designed for doing mathematics directly, not for formalising other theories and then working with them. However, in the same way as Basin and Howe showed that it is possible to use a 'foundational' theory such as *Nuprl* as an LF style framework, it should also be possible to use *PX* as a FS_0-style framework, using its

facilities for defining inductively defined classes. In fact in their book they develop a simple example meta-theorem: a tautology checker for propositional logic. But this is for propositional logic and does not need a binding mechanism — they do not discuss further applications, or how the application could be developed. One can argue against using PX rather than FS_0 as a framework in the same way one can argue against CC rather than LF: it is more complicated than necessary. Of course one could also reply to this by pointing out that stronger induction arguments are possible than in FS_0, but since few syntactic arguments (with the obvious exception of consistency proofs) need this, the extra power does does not, in practice, seem to be important.

The work of Sato has a different flavour from that of Hayashi and Nakano. In his work Sato presents a formal theory of s-expressions that is, again, suggestive of the theory FS_0, and argues for this as a general system for finite mathematics — an area in which he includes formal systems. In fact he explicitly relates his work to that of Smullyan, and it is very much in that tradition.

However, Sato does not discuss the practical issues related to doing mathematics or metamathematics in his framework. All the proofs that he presents are informal proofs about his theory (i.e., the same sort of proofs as Post or Smullyan would attempt). In fact, it is not clear how one could do general metamathematics (e.g., the normalisation theorem described here) in the theory since there is no induction available over the structure of a declared class. On the other hand, doing *mathematics* in a declared theory is clearly possible, and the facilities that he describes would provide a very powerful tactic language. He does not look at the issues that are raised by a full implementation of substitution; the substitution facility that he defines does not treat quantifiers, and so does not have to deal with problems such as, for example, alpha-conversion.

Perhaps the most interesting part of Sato's work, from the current perspective, is not so much that he presents a candidate for a framework theory, but that he uses it to develop a 'meta-circular' description for his theory in the same manner as McCarthy does for pure lisp in [58]. It would be interesting to see this developed further either in the direction of self-referential programming languages like Smith's 3-Lisp, or theoretical self-reference such as Gödel's work — the paper only goes as far as arguing that the definition of *eval* has the required properties.

§2 Metatheory as an end in itself: The work of Shankar

Shankar's doctoral work [67] was a proof, using the Boyer-Moore theorem prover (*Nqthm*), of the Church-Rosser theorem for the lambda-calculus. This is in some ways comparable with the work described above (the logic of *Nqthm* is, like FS_0, based on a theory of lists — in fact Feferman has suggested that a reworking of the same proof would be a good test of how FS_0 compares). However the work differs in other ways; Shankar is interested in how his proof improves understanding of the original informal proof, not in using it as a tool for reasoning in the lambda calculus.

The most impressive aspect is the scale of the theorem: it is a major result, and the proofs are long and difficult (the proof that he verifies is by Martin-Löf, and uses a lot of case analysis, which machines are good at). But while this is the most impressive, it is not the most relevant part of the work: he has to treat the same problem of how to handle bound variables as we do. After trying, and failing, to build the proof using a Church style, he adopted a de Bruijn style which he found more tractable. Afterwards, he was able to return to his original version and, using what he had learned, successfully prove parts of that. The same problems with bound variables were not encountered in the work presented here, and it is not reqally clear why this is so, but we believe that in part it was because we were able, in a natural way, to decompose the proof into two pieces, with the problems involving bound variables removed from the more complex parts of the proof, to be treated separately.

§3 Meta-theoretic extension

The work described in the last section is related, but not maybe the most relevant. A lot of work done on meta-theory as a tool to an end: to make a theorem proving system faster, or easier to use. The first paper on this was Davis and Schwartz [21], but this was speculative; the first practical work was by Boyer and Moore. This has since been built on by, for instance, workers in the Nuprl group and others.

3.1 Davis and Schwartz

Davis and Schwartz in [21] in 1979 suggested a completely general theorem proving system based on a powerful set theory, which would have a distinguished subtheory designed for doing metatheory, and the metatheoretic results could then be exploited to customise the environment of a PDS to be more suitable for particular purposes.

They argue, in the same way as here, that a meta-theoretic facility is necessary for a PDS that is able to follow the development of a proof as a mathematician would construct it. Further, that current PDSs, on the whole, did not allow the '[...] strong metamathematical component which is used to expand the rules of proof once theorems which (informally) justify such expansion have been proved'.

3.2 Meta-theoretic extension of the Boyer-Moore theorem prover

The first practical (and convincing) work done on meta-level extensions to a proof development system was by Boyer and Moore to their system *Nqthm*[17] which is a system for proving statements about a form of *Pure Lisp* (and therefore resembling, at least to some extent, FS_0). This work is described at length in [18]. *Nqthm* relies heavily on lemmas (these are provided as 'hints' about what is needed in the proof), rather than a tactic based interface: the user needs to provide a suitable set of these hints that the system can prove as a preparation for tackling the main theorem. This approach has been very effective in practice and has been used to generate formal proofs of very substantial theorems (like the Shankar work described above). The reason that a meta level facility was added to the logic is related closely to this approach to automated proof. The lemma mechanism that *Nqthm* is equipped with suffers from the typical drawbacks of PDSs (i.e., it does not allow schematic results to be used); a particular problem in a system so dependent on its lemma mechanism. The particular example they discuss is the problem of cancelling arbitrary terms in arithmetic; e.g.,

$$\frac{\Gamma \vdash b + c = a + j + k}{\Gamma \vdash b + c + i + x = (a + i + j) + k + x}. \tag{1}$$

This is an obvious and simple manipulation to do at the meta-level (simply make bags of each side of the expression, find the intersection, and delete that from both bags), and it is equally easy, at that level, to state that such a transformation is truth preserving. On the other hand, there is no way that the general principle can be stated as a lemma (in a form that the lemma mechanism can record).

This does not render their system unuseable, since it is always possible to identify the particular instances that are needed, and add them to the list of preparatory 'hints' to be proved individually. But it is certainly a difficulty, since the more instances that have to be explicitly supplied, the more work is involved for the user and the less claim there is to be doing *automatic* theorem proving.

To keep the lemma approach uniform, they suggest that it would be possible to extend the idea, so that instead of storing pieces of text it should be possible to store whole classes of wffs as lemmas — in other words, to use meta-theorems.

This means that it is necessary to install code so that the class can be recognised, and this, as they point out immediately, creates a serious problem: modifying the code of the theorem prover is a dangerous activity which could introduce inconsistency into the logic—they cite the semantic attachment facility in *FOL*[79] as a particular example of a system which runs this risk:

> 'Perilous acts, while perfectly legitimate in the hands of a careful implementor, are to be considered illegal in the hands of careless users.'

Conveniently though, *Nqthm* is itself written in *Lisp* so the problem is one of avoiding coding errors in *Lisp*; a problem that *Nqthm* is itself a perfect tool for solving.

So that it is possible to exploit the work done in the system to expand the theory, they do two things. First they define an isomorphism between their brand of *Pure Lisp* and a small subset of *Interlisp*; secondly, they add two new functions to the logic, `FORMP`, which is a test for well formedness, and `MEANING`, which is a function that takes a wff and an environment and returns the meaning of the wff in that environment (essentially a parametrised 'eval' function). These allow a user to prove that a rewrite function preserves meaning. Thus, if a meta level function called `CANCEL` is defined, to represent the class of rules of which (1) above is a particular instance, then if the theorem prover can demonstrate that:

```
(FORMP X) →
(MEANING X A) = (MEANING (CANCEL X) A) ∧ (FORMP (CANCEL X))
```

(where `A` is an arbitrary environment), then `CANCEL` translated into *Interlisp* can then be installed as a class of lemmas to be applied whenever an equivalence is found.

3.3 Meta-level facilities in Nuprl

Another substantial piece of work, quite closely related to the work described in the last section is that of Knoblock with the *Nuprl* PDS under Constable. This is documented in [48] and [49]. In fact in the first paper, two methods are described, but in the later thesis, this has been reduced to one, the other having apparently been abandoned (according to Howe in the in the first chapter of [44]) because the proof theory became just too difficult[ii].

The central idea that this work explored was that, since the logic that *Nuprl* implements corresponds closely to the lambda calculus, it should, in principle be possible to replace the informal programming language (meta-language) used for writing tactics in the system ML with a formal meta-theory which is simply another copy of the the object-level theory, only (conservatively) extended with facilities for reasoning about the structures of Nuprl style proofs. This work is described in the dissertation [49]: he describes his encoding and demonstrates how it is possible to synthesise tactics similar to those currently written in ML in the *Nuprl* programming language. This is, of course, not interesting by itself, since programming in *Nuprl* is more tricky than programming in ML, but he also examines the fundamental advantage of meta-level reasoning: that once a tactic has been formally verified, there is actually no reason to run it, it is enough to ensure that the 'pre' and 'post' conditions that define it are satisfied.

One interesting part to this work is that since he is interested in doing away completely with the ML programming language, he has to consider how to deal with the inevitable property of most tactics, that not only are they not guaranteed to succeed, but that intrinsically they cannot be guaranteed to succeed (what he terms 'search' tactics). These can be thought of as corresponding to meta-theorems with underspecified pre-conditions, or 'modal' meta-theorems, i.e., those that state that such a such a post condition *might* hold given such and such a pre-condition. This is a hard problem, and one that has been barely touched on. His suggestion is rather ad-hoc, he essentially gives the type of such a tactic as being crossed with a 'fail' type, so that if it does not succeed, the tactic will return something like the raised exception of ML. In this case of course, it is necessary actually to run the tactic that corresponds to the specification, and even the specification type is weak, since it can be satisfied by a function that always fails; there is no 'whenever possible' connective

to narrow it down.

Knoblock and Constable also suggest in [48] that it should be possible to get a partial reflection result for the *Nuprl* type theory, where reflection up to any particular universe level holds. This is an ambitious but reasonable result which corresponds to the results for Peano Arithmetic and Zermelo-Frænkel Set Theory: local reflection for any finitely axiomatisable sub theory of either is provable[iii]. Howe in his thesis suggests that this work has been abandoned, due to difficulties in getting formal proofs of the right results (because of the intractable nature of the proof theory), and certainly it does not feature in Knoblock's thesis except in the following:

> '...based on stratified internal evaluation functions. Although it contains some interesting aspects, it is deemed too complicated and cumbersome for practical use. [49]'

But, while this work was itself deemed to a dead end, it has laid the foundation for some of the current work on reflection at Cornell, which is discussed below.

3.4 Meta-level reasoning in the Calculus of Constructions

There has also been some work done in Edinburgh, both positive and negative, in the area of meta-level extensions to a declared theory by using results about the meta-theory to extend the object theory; this is documented in [52] and [76].

Pollack has produced in [52], as an example in the Lego system, a proof of the deduction theorem for a minimal logic declared in CC by using CC as a metalogic. Interestingly he has provided a proof that—for his formulation of the logic—LF is too weak to prove it (this is a particular instance of the statement, in [38], that induction over proofs is impossible in general).

Paul Taylor has also produced a short report [76], which expands this work with discussions about how to extend an algebraic theory (semi-groups) with new axiom schemas which correspond to a normal form tactic and an equivalence decision procedure; but this work has not been developed beyond one short note on a meta-theorem for algebras (this resembles, to an extent, both the work of Boyer and Moore described above, and that described below, from [44]).

§4 Conclusions

The work by Boyer and Moore treats essentially the same idea of meta-theory as a tool to an end as is discussed in the first part of this thesis. However the approach that they take is dependent very much on the contingency that the language of their theorem proving system is a fragment of the implementation language, also, the meta-level facilities they supply do not provide access to the full proof theory of the system. The work described here, on the other hand, allows arbitrary object theories, which can be extended with any result that is provable in primitive recursive arithmetic.

The Knoblock approach is slightly more distant, and complementary: our work discusses how to go about replacing tactics with meta-theorems, but does not deal with the sort of tactics with behaviour that cannot be satisfactorily formally specified. This necessarily includes the sort of powerful search tactics that are used to try to *automate*, as opposed to assist, proof construction.

§5 Work on reflection

With the idea of a framework theory in hand, the next step of exploring the possibilities offered by variations on the idea of reflective extension has been explored by various workers in the past (the investigations have even spilled over into the area of programming languages). This next section looks at ideas that are connected with the idea of reflection as it might be applied to a PDS, while more tangentially related work is discussed in the following section.

5.1 Weyhrauch's 'Prolegomena' and FOL

Weyhrauch's was one of the earliest investigations into self-reference in a theory and his ideas are developed in the very influential paper [79]. In this he outlines a PDS (called *FOL*) for doing mathematics in an arbitrary first order theory. The discussion in the paper is divided into two parts. The first outlines a concept of a *simulation structure*; this is a way to attach functions and values to symbols in the language, so that it is possible, instead of reasoning using the rules in the theory, to appeal to the semantics defined by the code instead. For instance, given a proposition $2+3=5$', it would be possible to appeal to the rules defining the connective '+', to prove that this equality is true; but it would also be possible to use the function *semantically attached* to '+', to add the values attached to '2' and '3' together directly. This is exactly

the sort of meta-level facility discussed in Abrahams' paper on the 'Proofchecker' program[1]. However, it is also the facility criticised by Boyer and Moore (discussed above) as dangerous in the hands of 'careless users'.

The second part of the paper is where the concept of two theory reflection rules are introduced (similar in some ways to the idea of meta-theoretic reasoning discussed in this thesis)

$$\frac{\Gamma \vdash_{T_1} \boldsymbol{A}}{\vdash_{T_2} Pr(\ulcorner \Gamma \vdash_{T_1} \boldsymbol{A} \urcorner)} \qquad \frac{\vdash_{T_2} Pr(\ulcorner \Gamma \vdash_{T_1} \boldsymbol{A} \urcorner)}{\Gamma \vdash_{T_1} \boldsymbol{A}}$$

in a theorem proving system as a way to extend its power by shortening proofs, and so that classes of lemmas (like Boyer and Moore discuss) are possible, instead of just single instances.

Finally, the concept of self reflection as a possible way to avoid the tower of meta theories by having a theory declared in the system that is a formal description of the system itself is introduced. This can raise problems: he points out the possibility of asking 'embarrassing questions' of the system when it is engaged in self-referential sorts of reasoning that do not give consistent answers. Interestingly, though, this is not regarded as a unqualified bug: it is suggested that these facilities will help a user to deal effectively with the problems that a non-monotonic, or even an inconsistent, logic might raise. Some of this work was suggested as part of papers that would appear in the future, but there has been little work published since on the subject.

The GETFOL project

More recently though, *FOL* has been given a new lease of life in research directed by Giunchiglia, which merges some of the ideas above, such as using the theorem prover to extend its own implementation with new code[32], [9], and also explores ideas in multilanguage systems such as Weyhrauch and Perlis [62], [63], suggest[31] (the subtitle of this significantly even evokes the subtitles of Perlis' papers on the subject).

5.2 Howe's work on reflection

Howe's work is reported initially in [44] where it makes up the third part of his thesis. Again, it is closely related, as a generalisation, to the work of Boyer and Moore reported in [18]. However, since he is working in *Nuprl*, he is able to take advantage of an enormously more powerful logic. The original reason for the work was as preparation for formalising Bishop's work on analysis, where a lot of results of an obviously metatheoretic nature occur. The example he cites is the theorem:

'if $f(x)$ is built only from x, $+$, $\times$, ... then f is continuous'

which has an obvious meta level character. Essentially, he develops a set of facilities for reflecting certain classes of algebraic expressions inside the theory and, with this, it is possible to make use of the same sorts of meta-theoretic results as Boyer and Moore achieve, only without going out of the logic.

Two functions are defined to enable this to be done

$$\mathit{Type} \in (\Pi a \in \mathit{Environment} \cdot (\mathit{Term}(a) \rightarrow \mathit{Set}))$$

and

$$\mathit{Val} \in (\Pi a \in \mathit{Environment} \cdot \Pi t \in \mathit{Term}(a) \cdot \mathit{Type}(a)(t))$$

where *Type* and *Val* correspond quite closely to the functions `FORMP` and `MEANING` in the Boyer-Moore work. The first thing to notice is that the versions here have an extra parameter, the environment, which allows a great deal more flexibility than in the earlier work, while keeping it as a conservative extension inside one system, unlike the Boyer-Moore work.

The system works as follows: if the current goal of the system is $\Gamma \vdash t$ then application of a special tactic `LiftUsing` with an environment σ generates a subgoal

$$\mathit{Val}(\sigma')(\Gamma),\ \sigma' : \Sigma, \ldots \vdash \mathit{Val}(\sigma')(\ulcorner t \urcorner)$$

(where Σ is the type of environments, and the ellipsis denotes the hypotheses that show that σ' is a well formed extension of σ that takes account of the variables in t); the other (well-formedness) subgoals will have been automatically disposed of. The second application of Val here reduces to a term equivalent to t, the change is that

now it is possible to apply functions to $\ulcorner t \urcorner$ in a way that allows Boyer-Moore style meta level reasoning, i.e., The proof obligations become

$$Type(\sigma)(\ulcorner t \urcorner) = Type(\sigma)(f(\ulcorner t \urcorner)) \ in \ Set$$

and

$$Val(\sigma)(\ulcorner t \urcorner) = Val(\sigma)(f(\ulcorner t \urcorner)) \ in \ Type(\sigma)(\ulcorner t \urcorner).$$

One significant point is the admission, after discussing using Nuprl in a way similar to *LF*, that:

> '... the reflection mechanism will encompass the *quantifier-free* portion of the logic'

in other words there are definite limits to the power that the approach gives.

5.3 Current work on reflection at Cornell

Current work on reflection at Cornell is exploring several directions. One of these is new work by Howe, which considers how it might be possible to reflect truth inside type theory. The other direction follows up on the work of Knoblock and Howe reported in [48], [49] and [44]. The most recent paper is [3] which is a second attempt to do what was suggested in [48]; i.e. to reflect, conservatively, the structure of Nuprl in itself.

The result mentioned in [48], that reflection up to some particular universe holds, is no longer an issue in this paper; instead a quite elegant theory of the structure of a proof system that uses indexed proof predicates, and this theory is shown to be a conservative extension of the proof structure of Nuprl.

How it is done

A conservative reflection rule is obtained here, instead of by stratifying the universe level, by stratifying the applications of the proof predicate. That is, the proof predicates are indexed over the natural numbers, and recognise only proofs that use proof predicates with a lower index than themselves. One of the minor points about this is that, apart from the hypothesis that there is a tactic that justifies the step (the first on the hypothesis list), there is no restriction on what is an acceptable rule in the subgoal deductions, i.e., the general form of a reflective inference rule is:

$$\frac{\vdash Pr_i\left(\ulcorner \frac{\Gamma_1 \vdash A_1 \quad \cdots \quad \Gamma_n \vdash A_n}{\Gamma \vdash A} \urcorner\right) \quad \Gamma_1 \vdash A_1 \quad \ldots \quad \Gamma_n \vdash A_n}{\Gamma \vdash A}$$

where $Pr_i(\cdot)$ is a proof predicate for proofs which exploit reflection at levels less than i. It is then possible to prove that this is conservative with respect to the the unextended theory, by induction on the index of the proof predicate.

Several applications for this facility are mentioned in the paper: the usual speed-up effects due to knowing that a tactic will succeed, polymorphism in theorems that make use of universe types, (interestingly) defining more efficient evaluation mechanisms for subclasses of expressions, e.g., by detecting when destructive updates are possible in the evaluation, etc.

5.4 Reflecting truth

Complementary to the notion of reflecting provability, is the notion of reflecting truth. Several significant papers have been published on this approach, mostly from a theoretical point of view.

Mendler and Aczel's LTC

In their paper [58], Mendler and Aczel consider several points. The title of the paper (*The notion of Framework and a framework for LTC*) would seem to suggest that it should be discussed above in the section on framework theories. However, they intend a different interpretation of the word 'framework' that is much closer to what is described earlier as a foundational theory. They suggest that a PDS should be developed from the begining with the idea that it should be 'open-ended': new ideas and new facilities should be easily added to the system.

They propose an example of such a system, based on Aczel's work on Frege structures [2]. The system is stratified, with the base level containing only predicate logic and simple ideas of arithmetic. Then each higher level of the system corresponds to the one below with its semantics reflected into itself. The limit of this is what they call (in their paper) T_4, with language L_4: the base language extended with the quoted syntax, and particularly the reflection predicates for all the layers of reflection below it. The major distinction between *LTC* and what is described in this work is obviously that they discuss reflecting the semantics of the logic, rather than the proof theory. But the resulting effect is similar, i.e., a stronger logic developed by applying a form of self-reference.

Feferman's work on truth

Feferman has recently published work[26] that stands quite close to that of Mendler and Aczel above. Unlike his earlier work on proof theoretic reflection principles and iterated (transfinite) extensions of theories, here he discusses the implications of reflecting truth, and avoiding progressions of theories.

The paper discusses two general mechanisms for creating the reflective closure of a theory using a truth predicate, and an unusual notion of a 'schematic theory', which is used to provide a particularly powerful form of such a closure. The second half then considers the strength of these extensions. The central result here is that that the schematic form of the reflective closure of PA is equivalent to the theory RA_{Γ_0} of ramified analysis with induction up to Γ_0, which is equivalent to predicative arithmetic. The paper also shows that the sorts of uniform (and therefore also local) reflection principles discussed above, in the chapter on the theory of reflection are provable in a logic extended this way.

5.5 Perlis' work on quotation

Finally, Perlis' work is also directly relevant: in two articles [62] and [63] he has looked at what is achieveable in first order logic given quotation facilities and minimal supporting facilities for substitution and the like. His work is intended as a reply to Montague [60], who argued that modal logics were necessary in order to deal with various intensional concepts.

Perlis deals mostly with problems related to first order doxastic and alethic logics (which makes it in some ways related to Feferman's work), but it also discusses the implications of machinery similar to what is proposed in previous chapters of this work, as a way of working with predicates that capture some notion of truth or belief.

5.6 Conclusion

The work on reflection in formal theories is clearly diverse. The work that is most closely related to what is described here is perhaps Weyhrauch's in that it discusses ideas related to both meta-theoretic extension (his notion of two theory reflection) and reflection in the sense that is discussed here (i.e., reflecting the idea of provability. However, he is also unconcerned with the soundness of his reflection principles, which

must count as a weakness, regardless of the gloss that he places on them. The other work that is clearly closely related is the Feferman approach of reflecting truth, which uses the same sort of 'style' of approach, though adding a truth predicate where we add a proof predicate. Feferman's work however is purely theoretical, whereas we are concerned with the practical issues involved (nevertheless we cannot help but admit that the results he achieves are dramatic and substantial).

§6 Other forms of meta-level and self-referential reasoning

As well as the work that is described in the sections above there is also a considerable amount of work that is less closely connected but, none the less, relevant.

6.1 Reflection in Lisp

Possibly the most interesting work that deals directly with the way that self reference can be exploited in a programming language, and certainly the most cited and influential, has been Smith's work on self-reference in (procedural) programming languages; specifically the language '3-lisp', which he designed, and which he describes in great detail, at great length, and with associated metaphysics, in his thesis [70] (a more concise description can be found in [71]).

Lisp is an unusual language, and one of its facilities from the beginning has been a facility for primitive self reference, in the form of `eval` and `funcall`, which allow self-referential, and self-modifying, code to be written[iv]. Smith's idea was to extend these primitive self-referential facilities to be more sensible. For instance, instead of the the standard notion of how a function application is executed (i.e., `LAMBDA`) a function application takes the current environment and continuation as well. Since these are made explicit in the system, the programmer is able to write functions that manipulate the environment itself, before passing the result on to an extended version of `EVAL`. This gives a programming language that has a great deal of (perhaps too much?) expressive power, and means that it is possible, in the same way as suggested in [16], to add various powerful control structures to the language. This could be considered the equivalent of the speedup results for formal theories. Of course, since a programming language such as *Lisp* is already 'Turing complete', extending it in such a way will not make it stronger in any formal sense. But Smith argues that once the reflection facilities have been exploited to design the new language facilities,

programs can be much shorter and more understandable than they otherwise would be.

A problem with Smith's approach is that perhaps it allows too much flexibility for a sensible programming language. In the light of this it could be argued that the logic programming approach is better, in that it places more restraints on what the programmer can do — whatever search strategy the programmer supplies to the proof predicate, it has, in the end, to produce a valid derivation; there are very few, if any, restrictions on what is allowed to a programmer in 3-Lisp.

6.2 Meta-level reasoning in Prolog

Work on meta-level reasoning that sits on the boundary between logic and programming languages is that by Bowen and Kowalski on how meta-level reasoning can be exploited in a programming language like *Prolog* [16]. They consider how to extend the facilities of a logic programming language by taking the usual language interpreter as the meta-level, and having an object-level interpreter $\texttt{Pr}(\cdot_1,\cdot_2)$ built on top of this. This way the object theory can be thought of as the query language for the data-base that is the *Prolog* system, while the meta-level is the language used to 'maintain' or modify the database, and even to modify such things as the search strategy.

The most interesting thing about their paper though, is the second part, where they attempt to 'amalgamate' the object and the meta language so that it is possible to communicate between the meta and object levels freely using rules like those that Weyhrauch suggests in [79]. Of course, since they are discussing logic programming, they restrict their theory to to the logic of pure *Prolog* rather than the whole of first order logic, and consistency problems are left to the user. The work is clearly related to, though not worked out in the same datail as, the work of Smith described above, and could be used for the same effects.

6.3 Conclusions

The work on reflection in programming languages nicely complements formal work on reflection. Clearly ideas on self reference in programming languages are being used, and therefore need to be treated formally. Further, Smith in particular argues (with convincing examples) that his informal reflection is a powerful way of improving the expressiveness of a programming language, which supports what is argued here, that reflection offers powerful ways of extending formal reasoning systems.

Notes

i. In fact, one of the reasons that Feferman gives explicitly in [25] for designing FS_0 in the form he describes it is to provide a tool that would allow a full formal proof of the second incompleteness theorem in a theory comparable to PRA

ii. It is interesting to note that there is a reference to what became [49] in the bibliography of [48] as 'Formal Metamathematics and Reflection in Type Theory' though it was later issued without any reference to reflection in the title.

iii. The result for PA can be found in [28] and for ZF in [60].

iv. This facility is not unique to *Lisp*; *COBOL* has also been equipped with a self-modification facility since its earliest days, and in comparison to that, even the facilities that the original *Lisp* 1.5 supplied are by no means primitive. However, *COBOL* programmers nowadays are very strongly discouraged from making any use of even this small facility, which remains in the standard for the sake of backwards compatibility and is not regarded by most people as A Good Thing [66].

Conclusions and Further Work

This thesis has presented a series of experiments to assess how practical it is to exploit meta-level techniques in mechanised formal reasoning. The thesis as a whole can be divided up into three parts: the experiments in meta-level reasoning, the experiments in reflection, and an assessment of the theory FS_0 in which the experiments were performed. These can be thought of as a common foundation and two 'annexes'. In this chapter I will summarise the work and discuss what can be concluded from each of these three parts. And finally, I consider what further work might be pursued.

§1 Meta-level reasoning

The purpose of this part was to examine how effective, and how practical, it is to construct meta-theorems that could be useful in the ordinary process of proof construction. I present three examples, constructed in different theories. First, the deduction theorem was proven in a small presentation of propositional logic, from Shoenfield; secondly, a version of the prenex normal form theorem was proven for a sequent calculus presentation of sorted first order logic; thirdly, the beta reduction rule for a presentation of the untyped lambda calculus is replaced with an equivalent which is explicitly recursive. The first two are the sorts of results that might be encountered in a logic textbook, while the third is perhaps more directly associated with practical problems of machine implementation of a theory.

1.1 The experiments

The deduction theorem

As an example of how a meta-level theorem can be used to reduce the amount of effort needed to construct a proof, the version of the deduction theorem presented here is by far the most dramatic. The theorem itself admits a simple constant time transformation of the goal. However if instead of using the theorem, an attempt is made actually to construct the corresponding proof, then in a reasonable practical instance literally thousands of proof steps will have to be taken, and the proof will grow to impractical proportions. Clearly meta-theory provides an enormous advantage in this case.

The prenex normal form theorem

This is an example of the sort of derived rule that a user might want to prove in practice: a syntactic transformation of a formula into a canonical form where it can be manipulated more easily and automatically, in a presentation (sequent calculus) that is likely to be encountered in practice. Here there is not the same sort of dramatic speedup over a tactic as in the deduction theorem above supplied, but still a substantial improvement is possible, since while the operation has the same complexity, it is by way of direct syntactic transformation, rather than as a side effect of complicated proof construction. Also, unlike with a tactic, the transformation

has a formal declarative description which can be used to combine it with other meta-theorems to construct other larger and more complex meta-theorems, and their associated functions.

The other feature of this experiment is that it gives an example of reasoning with formulae that involve bound variables where the binding mechanism is defined in 'classical' fashion (i.e., with distinctive names). This turns out not to be as difficult as perhaps might have been expected, although it was necessary to take some care so that general alpha conversions were not needed in the proof.

Beta reduction

In this experiment a declarative definition of the beta reduction rule for the simple lambda calculus is shown to be equivalent to a 'procedural' version which is explicitly recursive. One could think of the original version of the rule as a specification, and the replacement as an implementation, except that the former is equally useable as a rule, only it is not in a form where it is automatically disposable by a reasonable mechanism. Replacing it with a version that is explicitly recursive however, means that the validity of any rule applications become mechanically decidable immediately and thus much quicker. Since the two rules are extensionally identical though, they can be used interchangeably: for instance using the original for further meta-reasoning, but the new version for reasoning in the theory.

The second feature of this experiment is that it uses a 'de Bruijn' style binding mechanism, which contrasts directly with the version described above. This removes the problem of having to deal with alpha conversion, which is a possible advantage at the meta-level, but replaces it with the problem of a syntax that is very difficult for a human user at the object level.

1.2 Conclusions

Several possible advantages for meta-level reasoning were listed at the beginning of this thesis, and it is now possible to to make an assessment of how realistic these claims are in practice.

The first of these claims is that exploiting meta-theory can lead to an improvement in the usability, or flexibility, of a development system, and certainly the example of the deduction theorem is an example of this. Against the spectacular gains

here though, there are two possible caveats. It could be argued that Shoenfield's system is specially designed to be convenient for meta-theory at the cost of being completely intractable at the object theory, so it shows off this sort of manipulation to artificial advantage. Further, the meta-theory of propositional logic is particularly (unusually) simple — for instance it does not have bound variables — and so is not a good example of the general case. As a counter to this though, it can equally be pointed out that the definition of SP is easily extended to encompass all of classical predicate logic, a much more typical system. Then the fact that the raw system is practically unuseable becomes much less of an issue. Instead it would be possible, using the basic rules as a very simple foundation, to build different sorts of sets of rules for particular purposes, as necessary.

The other two experiments do not display the same improvement in performance: the function that is used to replace the tactic that would otherwise be used is of the same complexity. However, the replacing of a tactic with a simple syntactic transformation is a large improvement by itself: the fact that it is no longer necessary to verify each rule application means that the constant factor in the complexity of the operation is much reduced. Also these two illustrate well how with a meta-theorem the specification is made explicit, which helps a user to stay in intellectual control of the system. This is particularly the case in the example of the beta-reduction rule, where the declarative version of substitution is available for reasoning with, but for actually constructing deductions it is possible to use the (proven identical) procedural version.

§2 Reflection

The second set of experiments concentrate on how work that has been done on abstracting the behaviour of a Gödel style proof predicate can be exploited so that it is possible to use results in proof theory for extending a theory using reflection principles. The proof predicate that Gödel constructed for his original work is far too complex to be implemented for an arbitrary extension of primitive recursive arithmetic, and if this were the only proof predicate available then any results that involved it would be of purely theoretical interest. However, work has been done by Löb to abstract the important properties of such a predicate away from particular instantiations, as a set of derivability conditions.

First we described how the theory of sorted predicate logic, which was defined earlier for work on meta-theory, can easily be extended with quotation. Then a procedure for adding to the theory a predicate with properties based on the Löb derivability conditions was described. The conditions, as they are presented by Löb, are not quite adequate for our purposes: we are specifically interested in the possibilities that might be offered by uniform reflection, and this needs some way to do substitution into a quoted context. However it was possible to develop a suitable axiomatisation of substitution, and combine this with a slight extension of the Löb conditions, and this was used.

The question of whether it is possible to get the same sort of non-conservativity with this (*a priori* weaker) extension, than it is with a particular concrete instance of a proof, is then investigated. The experiments presented are for PRA and show that, here at least, it is possible to reproduce the non-conservativity results.

2.1 The examples presented

The three examples worked are all for primitive recursive arithmetic (i.e., elementary arithmetic with Σ^0_1-induction).

Course of values induction

The first example presented was a proof of a schema for course of values induction. Course of values induction over Σ^0_1-formulae is already available in PRA but the work needed to get it is not trivial (though the proof itself is not particularly difficult).

Ackermann's function

Unlike the example above, this was not a proof of a schema, but of a particular theorem, that Ackermann's function is everywhere defined. A proof of this is interesting here because it is the classic example of something that is known not to be provable in PRA.

The full induction schema

The final example presented here is a proof that the full induction schema of PA was provable in PRA extended with a single instance of the uniform reflection schema.

2.2 Conclusions

These three examples explore how it is possible to provide a facility for quotation, and from that, a general facility for using reflection to add reflexive extensions to a theory. The first and third examples, which describe proofs of schemas, are also interesting further examples of meta-theoretic reasoning techniques described above.

The point of the examples though, is of course to show that a theory extended in this way allows some of the same results that a Gödel proof predicate would, which it does. This is an interesting result if it scales up to larger theories. There are true propositions (without the artificial character of Gödel's sentence) that are not provable in, for instance PA, for instance termination of the Goodstein function, and there are many proofs that can be shortened directly by the use of more powerful proof principles. This approach would offer a way whereby theories could be extended using reflection (which is conceptually simple), without having to build the large technical apparatus that is normally associated with it.

This question is still open, so far as we know: it is not clear that proofs similar to those here cannot be used with (for instance) a schema for transfinite induction over ϵ_0. However we do not know. Finding out is clearly the next stage of work. If it does not work, then the question remains of why it works with theories of the strength of PRA?

§3 FS_0

Three distinctively different formal systems have been defined in FS_0 in the course of this work, and this has allowed some practical assessment of how it works in practice, as opposed to on the page. This assessment should be in the light of the features of the implementation itself though, so as a preliminary, this is considered.

3.1 The implementation

The implementation of FS_0 that was constructed for this work was based on a previous theorem proving system intended for a very different theory, and one problem is the direct result of this: the implementation is slow. This is simply a result of the fact that the original system was designed to make working directly in a large and complicated theory easy. The size of FS_0, as well as the purposes for which it is intended, on the other hand, make possible, and argue in favour of, a system optimised

for speed. Another problem was that as an initial implementation intended only for particular experiments, the system grew as an accretion of ad hoc patches as the need for various facilities became clear. In spite of this, though, the implementation was resilient enough for the purposes of the current experiments, and certainly provided a great deal of information about what would be needed in a 'production version'.

3.2 Using FS_0

Binding mechanisms

The most immediate point about FS_0 is that it does not have any facilities for handling bound variables directly, so any such mechanism has to be defined by the user. In practice this does not seem to be a problem; two different binding mechanisms are defined in the examples: a classical system for the language of sorted first order logic, and a de Bruijn style for the language of the lambda calculus. Since the binding mechanism has to be explicitly defined, in practice it is certainly going to be less efficient than what is possible with a framework such as LF, where it comes built in 'for free'. On the other hand, the fact that the binding mechanism is explicit means that reasoning with it directly and easily is possible, and this allows for easier meta-theory. Further, it is, with a little care possible to design a binding mechanism that can be shared by large collections of theories with languages represented in the similar manner.

Defining recursively enumberable sets

The purpose of FS_0 is to provide a theory for defining formal theories, and the primary mechanism that it provides for this is a facility for defining recursively enumerable classes. So the ease with which these can be defined is very important if the system is to be useable. The 'raw' form of FS_0, is not very useable, but given some sensible support software, as is described here, the situation is much improved, allowing a much clearer, and shorter notation. Given this support, for instance, formalising the theory SP takes only 40 lines from beginning to end (that is, for the basic theory itself: some further work has to be invested in building lemmas and the like to make it useable) which is a size comparable to the definition in LF.

Functions

The primary problem with FS_0 in practice is that it is much more dependent on functions for building the classes that define a theory than was expected. This causes a problem in two ways. The same criticisms can be made of the function facilities as for recursively enumerable classes, above. A compiler has been constructed for this system too, but the problems are more difficult to fix. The problem has two aspects.

First, the basic structures provided by the function definition facilities should correspond to what is needed in practice, and second, the function mechanism should be as efficient as possible. The first of these is a problem because the function definition mechanism only explicitly allows structural definition on s-expressions, while in practice function definitions tend to be at least defined on the structure of syntax encoded as s-expressions, which means course of values recursion on subtrees of the s-expressions. In fact this is often not really enough: witness the work that had to be done to construct a function that performed substitution in the beta-reduction rule, even though it was fairly easy to see that the function defined by the original equations must be terminating.

Second, since the only form of recursion available works directly on the structure of s-expressions, the natural form for the evaluation mechanism for function applications is very inefficient. This can be improved, to an extent, by having a lazy evaluation mechanism, but a recursion mechanism that allowed better evaluation strategies would be a better solution (this need not mean a stronger induction facility).

Facilities for meta-theory

The proof strength of FS_0 is equivalent to PRA, which makes it very weak compared to some theories which have been implemented on machines in the past. But in practice there is rarely going to be a need for more than this (one example class of exceptions is normalisation results for programming languages), and the induction facilities that are available are designed so as to be very suitable for the sort of 'applied' proof theory (i.e., for the purposes of augmenting the practical performance of a theorem prover) that is discussed here.

It should also be added here that, while FS_0 is a classical theory, in practice it was never necessary to use the rule of the excluded middle so all the proofs discussed

are constructive (while this was not exploited here, it is certainly something that might be made use of in a production system).

Notation

The final aspect of FS_0 worth noting is the problem of how to deal with terms in the encoded language. This has a problem that must be ubiquitous in framework theories: the natural syntax of encoded formulae in the framework theory usually bears no resemblance to the original, and is quite unreadable. Fixing this is simply a matter of providing some way of converting output into something more readable, but this is a problem that has not been addressed in more than a cursory manner.

§4 Summary

FS_0 was intended as a practical (as opposed to theoretical) Post style framework, and initial experience with it suggests that it is effective for that purpose, given that the implementation used was a prototype. A real 'production quality' implementation would of course be substantially different in appearance from the basic system, if only because of experience, but the the basic conception is sound.

The two sets of experiments above indicate that it is practical to exploit various types of meta-theoretic reasoning principles effectively in formal theorem proving, and further, that FS_0 is a practical foundation for this work.

§5 Further work

A great range of possible further work results from this thesis, so I will outline only some of them that I have specifically considered pursuing.

5.1 A reimplementation of FS_0

As a preamble to any further practical work, it will be necessary to build a new implementation of FS_0 which can incorporate what has been learned in this thesis. It will also be necessary to take account of the various pieces of further work outlined below.

In particular, the function definition mechanism needs to be considerably improved. Since *PRA* provides enough facilities to build anything that is ever going to be needed in practice (except interpreters — which can in fact be regarded, in a way, as reflection principles) ideally the improvements should be conservative, and there

are several candidate approaches that would satisfy such a requirement and provide a more efficient evaluation mechanism into the bargain.

5.2 Formal meta-theory

Further work in exploring meta-theory will clearly be of interest. While applied meta-theory is the area exploited here, Feferman has mentioned some applications of FS_0 which he feels would be particularly interesting, and a test of the theory at the same time.

> 'a formulation of Gödel's 2nd incompleteness theorem which applies to systems containing PRA in one way or another... Another candidate is the Gentzen cut-elimination theorem — or, assuming that — such things as the Herbrand-Gentzen midsequent theorem, or the interpolation theorem.'

Any of these would be demanding but rewarding results.

5.3 Schematic meta-theorems

The effort of constructing a definition of a theory, and then building theorems for that particular theory is substantial. However FS_0 is a second order theory, and this provides for the interesting prospect of schematic proofs.

It becomes easy to prove results that are quantified over classes of theories. For instance, it is possible not only to build a 'skeleton' definition of a class of theories (such as, first order logic), and then quantify over it, but it is also possible to prove theorems about that 'skeleton' theory which then apply to any instantiation. Consider the definition of the language $\mathcal{L}$ and the associated theory of predicate logic that is presented in this thesis. It is possible to imagine proofs of propositions of the form

$$\forall R \forall a(a \in wff_C \wedge a \in P \rightarrow a \in T\langle R \rangle)$$

which, once proven, would be a result useable with any theory defined as an extension of sorted first order logic (in other words, a derived rule). This means that it would be possible to exploit one meta-theoretic result over a variety of theories without having to redo the work each time.

5.4 Reflection principles

There are a collection of points that should be investigated here, both theoretical and practical. The relationship between abstract proof predicates defined by the derivability conditions on the possibilities for a concrete proof predicate, and concrete instances of the proof predicate itself is not clear. See work done in Russia [54], which has only recently been published in English, on investigating the modal interpretation of provability, which extends Solovay's completeness theorem to the predicate form of provability logic for instance.

Also, in his recent survey paper on reflection, Feferman's results for reflecting the truth of propositions instead of provability, suggest that that this would be an interesting alternative approach (in fact there is an implication in the paper that FS_0 would be a suitable vehicle for developing this work in practice).

Appendix: Course of values induction for FS_0

In practice, the primitive stepwise induction facility of FS_0 is not as useful as course of values induction on the ordering 'is a subtree of'. So I will present a proof here of induction for that. The proof is also a nice illustration of how flexible FS_0 is, compared to, say, PRA. In fact there are two points worth making: first, because lists are primitive, the proof is quite easy to build from basic components of the system, whereas in a system like PRA there would have to be several chapters of development work first, designing Gödel numbering schemes etc. Second, whereas in a system like PRA course of values induction is a schema, and so is not provable once and for all as a lemma, in FS_0 because it is a second order theory, it is possible to prove a single lemma quantified over classes.

Most of the proof is trivial, but quite long, compared to the other results here (even if short compared to the equivalent in PRA), so I will just sketch it.

The first step is to define the partial ordering relation $\prec$, and this is done using the following definitions

$$
\begin{aligned}
if_F &\triangleq \mathcal{C}[D, \mathcal{P}[\pi_1, K_{true}, \pi_{12}, \pi_{22}]] \\
or_F &\triangleq \mathcal{C}[if_F, \mathcal{P}[\pi_1, K_{true}, \pi_2]] \\
equal_F &\triangleq \mathcal{C}[D, \mathcal{P}[\pi_1, \pi_2, K_{true}, K_{false}]] \\
f_\prec &\triangleq \mathcal{C}[if_F, \\
&\qquad \mathcal{P}[\mathcal{C}[equal_F, \mathcal{P}[\pi_{1111}, \pi_{2111}]] \\
&\qquad\quad K_{true}, \\
&\qquad\quad \mathcal{C}[if_F, \\
&\qquad\qquad \mathcal{P}[\mathcal{C}[equal_F, \mathcal{P}[\pi_{1111}, \pi_{211}]], \\
&\qquad\qquad\quad K_{true}, \\
&\qquad\qquad\quad \mathcal{C}[or_F, \mathcal{P}[\pi_{12}, \pi_2]]]]]] \\
f_\prec &\triangleq \mathcal{R}(K_{false},)
\end{aligned}
$$

and

$$S_1 \prec S_2 \equiv f_\prec(S_1, S_2) = true$$

Then the problem is to show

$$\vdash \forall C \forall x(\forall y(y \prec x \rightarrow y \in C) \rightarrow x \in C) \rightarrow \forall x(x \in C)$$

This reduces quickly to

$$\forall x(\forall y(y \prec x \rightarrow y \in C) \rightarrow x \in C) \vdash \forall x(x \in C)$$

Then given the lemma

$$\forall x(\forall y(y \prec x \rightarrow y \in C) \rightarrow x \in C) \vdash \forall x \forall y(y \prec x \rightarrow y \in C)$$

the result follows by the usual proof. Thus the problem has been reduced to the lemma. But the usual proof of the lemma follows by induction on x, which is not possible here since it is not over a Σ_1 formula. It is necessary to find a class C' having the property that

$$\vdash x \in C' \leftrightarrow \forall y(y \prec x \rightarrow y \in C). \qquad (*)$$

Then it is possible to prove a modified form of the lemma,

$$\forall x(\forall y(y \prec x \rightarrow y \in C) \rightarrow x \in C) \vdash \forall x(x \in C')$$

by induction on x. The only problem is, of course, in building the class C'.

The definition of C'

The first stage is the building the class C'' which is defined as the class consisting of the base case

$$(O, O) \in C''$$

closed under the rule

$$\frac{(S_1, S_2) \in C'' \qquad (S_3, S_4) \in C''}{((S_1, S_2), (S_3, S_4), (S_2, S_4)) \in C''} \; S_2 \in C \wedge S_4 \in C$$

So C' can be defined simply as

$$x \in C' \leftrightarrow \exists y\,((y, x) \in C'')$$

Now all that is left is to prove $(*)$

Right to left

We first show

$$\forall x(x \prec y \rightarrow x \in C) \vdash \forall x(x \prec y \rightarrow x \in C'). \qquad (L1)$$

This is proven by simple induction. Since $x \prec y$ is defined to be $f_{\prec}(x,y) = true$ it is possible to define the complement, $x \not\prec y$ as $f_{\prec}(x,y) \neq true$, and make use of the identity

$$\vdash (x \prec y \rightarrow x \in C') \leftrightarrow (x \not\prec y \vee (x \prec y \wedge x \in C')).$$

it is also possible to construct the comprehension of the right hand side for x, (call it Co), the goal can be replaced with that, and then it is possible to use induction. In the base case, $O \in Co$ since $(O,O) \in C''$ and therefore $\exists y((y,O) \in C'')$. In the step case, the goal can be reduced to

$$\begin{array}{l} \forall x(x \prec y \rightarrow x \in C), \\ a \prec y \rightarrow a \in C', \\ b \prec y \rightarrow b \in C', \\ (a,b) \prec y \\ \quad \vdash (a,b) \in C' \end{array}$$

which, since $a \prec y$ and $b \prec y$ by transitivity of $\prec$ reduces to

$$\begin{array}{l} \forall x(x \prec y \rightarrow x \in C), \\ a \in C', b \in C', (a,b) \prec y \\ \quad \vdash (a,b) \in C' \end{array}$$

Then, by applying the first hypothesis to a and b, and the definition of C'

$$\begin{array}{l} a \in C, a \in C, \\ \exists z\,((z,a) \in C''), \exists z\,((z,b) \in C'') \\ \quad \vdash \exists z\,((z,(a,b)) \in C'') \end{array}$$

Removing the existentials in the hypothesis list then gives

$$\begin{array}{l} a \in C, a \in C, \\ (z',a) \in C'', (z'',b) \in C'' \\ \quad \vdash \exists z\,((z,(a,b)) \in C'') \end{array}$$

and from the definition of C'' it is easy to show that

$$a \in C, a \in C,$$
$$(z', a) \in C''), (z'', b) \in C''$$
$$\vdash ((z', a), (z'', a), (a, b)) \in C''$$

At this point it is possible to return to the point of this section, which is to prove the right to left case of $(*)$, i.e, (by the definition of C' and some simplification) that

$$\forall y(y \prec x \rightarrow y \in C) \vdash \exists z\,((z, x) \in C)$$

By cutting in the result of $(L1)$, we have

$$\forall y(y \prec x \rightarrow y \in C), \forall y(y \prec x \rightarrow y \in C') \vdash \exists z\,((z, x) \in C'')$$

and, by an a analysis by cases on $x = O \vee \exists y\,\exists z\,(x = (y, z))$ the result follows easily.

Left to right

Having outlined the sort of techniques that are used, this section will be much shorter. To recap, the required result is that

$$\vdash y \in C' \rightarrow \forall x(x \prec y \rightarrow x \in C).$$

This follows from

$$\vdash y \in C'' \rightarrow \forall x(x \prec \pi_2 y \rightarrow x \in C \wedge x \in C')$$

which follows by induction on the structure of C''.

Appendix: the rules of $\mathcal{L}$

$$\frac{}{\Gamma \vdash \boldsymbol{A}}\ \text{basic} \qquad \text{where} \qquad \boldsymbol{A} \in \Gamma$$

$$\frac{}{\Gamma \vdash \boldsymbol{A} \vee \boldsymbol{A} \to \bot}\ \text{exmid}$$

$$\frac{\Gamma \vdash \boldsymbol{A} \qquad \Delta \vdash \boldsymbol{B}}{\mathrm{E} \vdash \boldsymbol{B}}\ \text{cut} \qquad \text{where} \qquad \Gamma \subset \mathrm{E}$$

$$\Delta \subset \mathrm{E} \cup \{\boldsymbol{A}\}$$

$$\frac{\Delta \vdash \boldsymbol{C}}{\Gamma \vdash \boldsymbol{C}}\ \text{thin} \qquad \text{where} \qquad \Delta \subset \Gamma$$

$$\frac{\Delta \vdash \bot}{\Gamma \vdash \boldsymbol{C}}\ \bot\text{-r} \qquad \text{where} \qquad \Delta \subset \Gamma$$

$$\frac{\Delta \vdash \boldsymbol{A}}{\Gamma \vdash \boldsymbol{A} \vee \boldsymbol{B}}\ \vee\text{-r}_1 \qquad \text{where} \qquad \Delta \subset \Gamma$$

$$\frac{\Delta \vdash \boldsymbol{B}}{\Gamma \vdash \boldsymbol{A} \vee \boldsymbol{B}}\ \vee\text{-r}_2 \qquad \text{where} \qquad \Delta \subset \Gamma$$

$$\frac{\Gamma \vdash \boldsymbol{A} \qquad \Delta \vdash \boldsymbol{B}}{\mathrm{E} \vdash \boldsymbol{A} \wedge \boldsymbol{B}}\ \wedge\text{-r} \qquad \text{where} \qquad \Gamma \subset \mathrm{E}$$

$$\Delta \subset \mathrm{E}$$

$$\frac{\Delta \vdash \boldsymbol{B}}{\Gamma \vdash \boldsymbol{A} \to \boldsymbol{B}} \to\text{-r} \quad \text{where} \quad \Delta \subset \{\boldsymbol{A}\} \cup \Gamma$$

$$\frac{\Gamma \vdash \boldsymbol{C} \qquad \Delta \vdash \boldsymbol{C}}{\mathrm{E},\ \boldsymbol{A} \vee \boldsymbol{B} \vdash \boldsymbol{C}} \vee\text{-l} \quad \text{where} \quad \Gamma \subset \{\boldsymbol{A}\} \cup \mathrm{E}$$

$$\Delta \subset \{\boldsymbol{B}\} \cup \mathrm{E}$$

$$\frac{\Delta \vdash \boldsymbol{C}}{\Gamma,\ \boldsymbol{A} \wedge \boldsymbol{B} \vdash \boldsymbol{C}} \wedge\text{-l} \quad \text{where} \quad \Delta \subset \{\boldsymbol{A}\} \cup \{\boldsymbol{B}\} \cup \Gamma$$

$$\frac{\Gamma \vdash \boldsymbol{A} \qquad \Delta \vdash \boldsymbol{C}}{\mathrm{E},\ \boldsymbol{A} \to \boldsymbol{B} \vdash \boldsymbol{C}} \to\text{-l} \quad \text{where} \quad \Gamma \subset \mathrm{E}$$

$$\Delta \subset \{\boldsymbol{B}\} \cup \mathrm{E}$$

$$\frac{\Gamma \vdash \boldsymbol{A}[v/v']}{\Delta \vdash \forall v' \boldsymbol{A}} \forall\text{-r} \quad \text{where} \quad \Gamma \subset \Delta$$

$$v \notin \Gamma$$

$$\frac{\Gamma \vdash \boldsymbol{A}[t/v]}{\Delta \vdash \exists v\, \boldsymbol{A}} \exists\text{-r} \quad \text{where} \quad \Gamma \subset \Delta$$

$$t \in \mathit{term}_C$$

$$\frac{\Gamma,\ \boldsymbol{A}[t/v] \vdash \boldsymbol{C}}{\Delta,\ \forall v \boldsymbol{A} \vdash \boldsymbol{C}} \forall\text{-l} \quad \text{where} \quad \Gamma \subset \Delta$$

$$t \in \mathit{term}_C$$

$$\frac{\Gamma,\ \boldsymbol{A}[v/v'] \vdash \boldsymbol{C}}{\Delta,\ \exists v'\, \boldsymbol{A} \vdash \boldsymbol{C}} \exists\text{-l} \quad \text{where} \quad \Gamma \subset \Delta$$

$$v \notin \Gamma$$

Appendix: The construction of cro_F and $lift_F$

Constructing cro_F

a solution for cro_F is slightly tricky, or more accurately, messy. It cannot be defined easily at the same level of abstraction as above. Instead it is necessary to go inside the abstract representation of lists to do it. The specific mechanics is as follows. hd, tl and list allows the use of apparently infinite lists that default to a particular value, so that a list $a = (a_1, \ldots, a_n, a_{n+1}, a_{n+2}, \ldots)$, where $a_{n+j} = a_n$, is held in a structure $(a_n, (a_1, \ldots a_{n+j-1}))$ where, usually, $j = 0$. cro_F should have the behaviour

$$cro_F(a, b) = ((a_n, b_m), ((a_1, b_1), \ldots (a_{p-1}, b_{p-1})))$$

$(p \geq max(m, n))$. So cro_F is defined to be

$$cro_F((x, x'), (y, y')) \triangleq ((x, y), cro_F{}'(x, y, x', y'))$$

where $cro_F{}'$ has the properties

$$len_F(cro_F{}'((x, x'), (y, y'))) \not< len_F(x') \qquad p6$$

$$len_F(cro_F{}'((x, x'), (y, y'))) \not< len_F(y') \qquad p7$$

$$i < len_F(cro_F{}'((x, x'), (y, y'))) \rightarrow cro_F{}'((x, x'), (y, y'))_i = ((x, x')^i, (y, y')^i) \qquad p8$$

Then $p4$ can be verified against this as follows

$$\begin{aligned} & cro_F((x, x'), (y, y'))^i \\ & \quad = ((x, y), cro_F{}'((x, x'), (y, y'))^i \\ & \quad = \begin{cases} cro_F{}'((x, x'), (y, y'))_i & \text{if } i < len_F(cro_F{}'((x, x'), (y, y'))) \\ (x, y) & \text{if } i \not< len_F(cro_F{}'((x, x'), (y, y'))) \end{cases} \qquad \text{by } D1 \end{aligned}$$

The first case follows trivially, by $p8$, while the second follows by

$$\begin{aligned} & i \not< len_F(cro_F{}'((x, x'), (y, y'))) \\ & \qquad \rightarrow i \not< len_F(x') \end{aligned}$$

by transitivity of $\not<$, $p6$,and $p8$

$$\rightarrow (x, x')^i = x \qquad \text{by } D1$$

The result for (y, y') follows almost identically, and completes the proof.

Then, if $cro_F{}'$ is defined

$$cro_F{}'(x,y) \triangleq \begin{cases} cro_F{}''(len_F(\pi_1 y), x, y) & \text{if } len_F(\pi_2 x) < len_F(\pi_2 y) \\ cro_F{}''(len_F(\pi_1 x), x, y) & \text{if } len_F(\pi_2 x) \not< len_F(\pi_2 y) \end{cases}$$

and

$$cro_F{}''(l,a,b) \triangleq \text{map}\langle f\rangle(count(l),(a,b))$$
$$\text{where } f(x,(y,z)) = (y^x, z^x)$$

These can be verified against $p6$–$p8$ as follows. First $p6$, which is proven by case analysis on

$$len_F(x') < len_F(y') \vee len_F(x') \not< len_F(y')$$

Assuming the left disjunct, i.e., $len_F(x') < len_F(y')$, then

$$len_F(cro_F{}'((x,x'),(y,y'))) \not< len_F(x') \leftrightarrow len_F(x') < len_F(cro_F{}'((x,x'),(y,y')))$$

and

$$len_F(cro_F{}'((x,x'),(y,y')))$$
$$= len_F(cro_F{}''(len_F(y'),(x,x'),(y,y')))$$

by the definition of $cro_F{}'$ and the hypothesis

$$= len_F(\text{map}\langle f\rangle(count(len_F(y')),((x,x'),(y,y'))))$$

by definition of $cro_F{}''$

$$= len_F(count(len_F(y'))) \qquad \text{by } D2$$
$$= len_F(y') \qquad \text{by } D3$$

so

$$len_F(x') < len_F(cro_F{}'((x,x'),(y,y')))$$
$$\leftrightarrow len_F(x') < len_F(y')$$

which follows from the hypothesis. The other disjunct is much the same, as is the proof of $p7$

The verification of $p8$ can use the same case analysis, so, again first assuming $len_F(x') < len_F(y')$, the proof is as follows

$$i < len_F(cro_F{}'((x,x'),(y,y'))) \rightarrow cro_F{}'((x,x'),(y,y'))_i = ((x,x')^i,(y,y')^i)$$

can be reduced to

$$i < len_F(y') \rightarrow \text{map}\langle f\rangle(count(len_F(y')),((x,x'),(y,y')))_i = ((x,x')^{\overline{i}},(y,y')^i)$$

but

$$i < len_F(y') \rightarrow \text{map}\langle f\rangle(y', ((x,x'),(y,y')))_i = f(y'_i, ((x,x'),(y,y'))) \qquad \textit{by } D4$$

and

$$len_F(y') = len_F(count(len_F(y'))) \qquad \textit{by } D3$$

so the problem is reduced to

$$\begin{aligned} i < len_F(y') &\rightarrow f(count(len_F(y'))_i, ((x,x'),(y,y'))) = ((x,x')^i, (y,y')^i) \\ &\rightarrow f(i, ((x,x'),(y,y'))) = ((x,x')^i, (y,y')^i) \end{aligned}$$

since by D5, $i < y \rightarrow count(y)_i = i$, and this in turn gives

$$\rightarrow ((x,x')^i, (y,y')^i) = ((x,x')^i, (y,y')^i)$$

(by the definition of f given in cro_F''), giving the result.

Constructing $lift_F$

The solution to $lift_F$ that satisfies $p5$ constructed in a way that is very similar to the solution for sub_F. $lift_F$ is defined as:

$$lift_F(x,y) = lift_F'(x,y)^O$$

then $p5$ follows from

$$(y,w,j,y') \in lift_C \rightarrow y' = lift_F'(y,w)^j$$

and the rest of the construction follows the proof of sub_F', only starting from

$$\begin{aligned} lift_F'((\text{'var'}, i), j)^k &= \begin{cases} (\text{'var'}, i) & \text{if } i < k \\ (\text{'var'}, i+j) & \text{if } i \not< k \end{cases} \\ lift_F'((\text{'}\lambda\text{'}, M), x)^k &= (\text{'}\lambda\text{'}, (lift_F'(M,x)^{k+1})) \\ lift_F'((\text{'app'}, (M,N)), x)^k &= (\text{'app'}, cro_F(lift_F'(M,x), lift_F'(N,x))^k) \end{aligned}$$

and the function *list* is replaced with

$$list'(x,y,j)^k = \begin{cases} x & \text{if } j < k \\ y & \text{if } j \not< k \end{cases}$$

Bibiliography

1. P. Abrahams, *Application of Lisp to Checking Mathematical Proofs*, in "The Programming Language *Lisp*: its Operations and Applications", E. Berkeley and D. Bobrow eds., M.I.T. Press, Cambridge, Massachusetts, (1964).
2. P. Aczel, *Frege Structures and the notions of proposition, truth and set*, in "The Kleene Symposium", J. Barwise, H. Keisler and K. Kunen eds., North Holland, Amsterdam, (1978), pages 31–59.
3. S. Allen, R. Constable, D. Howe and W. Aitken, *The Semantics of Reflected Proof*, Department of Computer Science (preprint), Cornell University, Ithaca, New York, (1990).
4. A. Avron, *Simple Consequence Relations (ECS-LFCS-87-30)*, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh, (1987).
5. A. Avron, F. Honsell and I. Mason, *Using Typed Lambda Calculus to Implement Formal Systems on a Machine (ECS-LFCS-87-31)*, Laboratory for The Foundations of Computer Science, Department of Computer Science, University of Edinburgh, (1987).
6. H. Barendregt, "The Lambda Calculus, Its Syntax and Semantics (revised edition)," North-Holland, Amsterdam, (1984).
7. D. Basin, *Building Problem Solving Environments in Constructive Type Theory (Thesis, TR 89–1063)*, Department of Computer Science, Cornell University, Ithaca, New York, (December 1989).
8. D. Basin and R. Constable, *Metalogical Frameworks*, in Proceedings of the Second Workshop on Logical Frameworks, informal, distributed by e-mail.
9. D. Basin, F. Giunchiglia and P. Traverso, *Automating Meta-Theory Creation and System Extension*, in Proceedings of the Second Congress of the Italian Association for Artificial Intelligence, Springer, Berlin, (October 1991), pages 48–57.
10. D. Basin and D. Howe, *Some Normalisation Properties of Martin-Löf's Type Theory*, in Theoretical Aspects of Computer Software, Springer, Berlin, (1991).
11. N. de Bruijn, *Lambda Calculus notation with nameless dummies, a tool for automatic formula manipulation*, Indagationes Mathematicae, vol. 34, (1972), pages 381–392.
12. N. de Bruijn, *A Survey of the Project AUTOMATH, Its Usage and Some of Its Extensions*, in "To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism", J. Seldin and J. Hindley eds., Academic Press, New York, (1980), pages 589–606.
13. A. Bundy, F. van Harmelen, C. Horn and A. Smaill, *The Oyster-Clam system*, in 10th International Congerence on Automated Deduction (Lecture Notes in Artificial Intelligence No. 449), Springer, Berlin, (1990).
14. A. Church, "The Calculi of Lambda Conversion," Princeton University Press, Princeton, (1941).
15. A. Church, *A formulation of the simple theory of types*, Journal of Symbolic Logic, vol. 5, (1940), pages 56–68.
16. K. Bowen and R. Kowalski, *Amalgamating Language and Metalanguage in Logic Programming*, in "Logic Programming", K. Clark et. al. eds., Academic Press, London, (1982), pages 153–172.
17. R. Boyer and J. Moore, "A Computational Logic," Academic Press, New York, (1981).
18. R. Boyer and J. Moore, *Metafunctions: proving them correct and using them efficiently as new proof procedures*, in "The Correctness Problem in Computer Science", R. Boyer and J. Moore eds., Academic Press, New York, (1981), pages 103–184.

19. R. Constable et. al., "Implementing Mathematics with the Nuprl Proof Development System," Prentice-Hall, Englewood Cliffs, New Jersey, U.S.A., (1986).
20. D. van Daalen, "The Language Theory of Automath," Technische Hogeschool Eindhoven, Eindhoven, (1980).
21. M. Davis and J. Schwartz, *Metamathematical Extensibility for Theorem Verifiers and Proof Checkers (Courant Computer Science Report No. 12)*, Courant Institute of Mathematical Sciences, New York, (1977).
22. A. Ehrenfeucht and J. Mycielski, *Abbreviating Proofs by Adding New Axioms*, Bulletin of the American Mathematical Society, vol. 77, (1971), pages 366–367.
23. S. Feferman, *Transfinite Recursive Progressions of Axiomatic theories*, Journal of Symbolic Logic, vol. 27, (1962), pages 259–316.
24. S. Feferman and W. Sieg, *Proof Theoretic Equivalences between Classical and Constructive Theories for Analysis*, in "Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof Theoretic Studies' (Lecture notes in Mathematics, No. 897)", W. Buchholz, S. Feferman, W. Pohlers and W. Sieg, eds., Springer, Berlin, (1981), pages 78–142.
25. S. Feferman, *Finitary Inductively Presented Logics*, in Logic Colloquium '88, North-Holland, Amsterdam, (1990).
26. S. Feferman, *Reflecting on Completeness*, Journal of Symbolic Logic, vol. 56, (1991), pages 1–49.
27. S. Feferman, Personal Communication, (1991).
28. J. Y. Girard, "Proof Theory and Logical Complexity, Vol. I," Bibliopolis, Naples, (1987).
29. A. Fraenkel, Y. Bar-Hillel and A. Levy, "Foundations of Set Theory," North-Holland, Amsterdam, (1973).
30. J. Y. Girard, Y. Lafont and P. Taylor, "Proofs and Types," Cambridge University Press, Cambridge, England, (1989).
31. F. Giunchiglia and L. Serafini, *Multilanguage Hierarchical Logics (or: how we can do without modal logics)*, Mechanised Reasoning Group, I.R.S.T., Trento, Italy, (unpublished draft).
32. F. Giunchiglia and P. Traverso, *Reflective Reasoning With and Between a Declarative Metatheory and the Implementation Code*, in Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence, Inc., California.
33. M. Gordon, R. Milner and C. Wadsworth, "Edinburgh LCF: a mechanised logic of Computation (Lecture notes in Computer Science, No. 78)," Springer, Berlin, (1979).
34. D. Gries, "The Science of Programming," Springer, New York, (1981).
35. K. Gödel, *Über die Lange der Beweis*, Ergenis eines mathematischer Kolloquiums, vol. 7, pages 23–24.
36. K. Gödel, *Über formal unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme*, Monatshefte für Mathematik und Physik, vol. 38, (1931), pages 173–198.
37. F. van Harmelen, A. Simpson, F. Giunchiglia, L. Serafini and A. Smaill, *A discussion about naming relations (RFL/ViA/I.4/1)*, Esprit Basic Research Project P3178 Reflect, S.W.I., University of Amsterdam, (July, 1990).
38. R. Harper, F. Honsell and G. Plotkin, *A Framework for Defining Logics*, in Second Symposium on Logic in Computer Science, Computer Society Press, (June, 1987).
39. R. Harper, F. Honsell and G. Plotkin, *A Framework for Defining Logics*, Journal of the Association for Computing Machinery, (To appear).

40. S. Hayashi and H. Nakano, "PX: A Computational Logic," M.I.T. Press, Cambridge, Massachusetts, (1988).
41. I. Hayes, "Specification Case Studies," Prentice-Hall International, London, (1987).
42. D. Hilbert, *Neubegründung der Mathematik*, in "Abhandlungen aus dem mathematischen Seminar der Hamburgischen Universität, Band I", Universität Hamburg, Hamburg, Germany, (1922).
43. D. Hilbert and P. Bernays, "Grundlagen der Mathematik, Bande II," Springer, Berlin, (1939).
44. D. Howe, *Automating Reasoning in an Implementation of Constructive Type Theory (Thesis, TR 88-925)*, Department of Computer Science, Cornell University, Ithaca, New York, (June 1988).
45. G. Hughes and M. Creswell, "An Introduction to Modal Logic," Methuen, New York, (1968).
46. L. S. Jutting, *Checking Landau's Grundlagen in the Automath System (Thesis)*, Eindhoven University, Netherlands, (1977).
47. S. Kleene, "Introduction to Metamathematics," North-Holland, Amsterdam, (1952).
48. T. Knoblock and R. Constable, *Formalised Metareasoning in Type Theory (TR-86-742)*, Department of Computer Science, Cornell University, Ithaca, New York, (March 1986).
49. T. Knoblock, *Metamathematical Extensibility in Type Theory (Thesis, TR-87-892)*, Department of Computer Science, Cornell University, Ithaca, New York, (December 1987).
50. G. Kreisel and A. Lévy, *Reflection Principles and their use for establishing the Complexity of Axiomatic Systems*, Zeitschrift Für Mathematische Logik, und Grundlagen der Mathematik, vol. 14, (1968), pages 97–142.
51. M. Löb, *Solution of a problem of Leon Henkin*, Journal of Symbolic Logic, vol. 20, (1955), pages 115–118.
52. Z. Luo, R. Pollack and P. Taylor, *How to use LEGO (A preliminary User's Manual)*, Laboratory for the Foundations of Computer Science, (Unpublished), Department of Computer Science, University of Edinburgh, (October, 1989).
53. W. McCune, *The Otter User's Guide*, Argonne National Laboratory, Illinois, (1989).
54. V. McGee, *A Survey of Results by Artemov and Vardanyan on the Modal Logic of Provability*, Journal of Symbolic Logic, vol. 56, (1991), pages 329–332.
55. P. Madden, *Automated Program Transformation Through Proof Transformation (Thesis)*, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, (1991).
56. Z. Manna, "Mathematical Theory of Computation," McGraw-Hill, Tokyo, (1974).
57. P. Martin-Löf, "Intutitionistic Type Theory," Bibliopolis, Naples, (1984).
58. P. Mendler and P. Aczel, *The notion of a Framework and a framework for LTC*, in Third Annual Symposium on Logic in Computer Science, Computer Society Press, Massachusetts Avenue, Washington D.C., (1988).
59. G. Minc, *Quantifier-free and one-quantifier systems*, Journal of Soviet Mathematics, vol. 1, (1973), pages 71–84.
60. R. Montague, *Syntactical Treatments of Modality, with Corollaries on Reflexion Principles and Finite Axiomatisability*, Acta Philosophica Fennica, vol. 16, (1963), pages 153–167.
61. C. Murthy, *Extracting Constructive Content in Classical Proofs (Thesis)*, Department of Computer Science, Cornell University, Ithaca, New York, (December 1990).
62. D. Perlis, *Languages with Self-Reference I: Foundations (or: We Can Have Everything in First-Order Logic)*, Artificial Intelligence, vol. 25, (1985), pages 301–322.

63. D. Perlis, *Languages with Self-Reference II: Knowledge, Belief, and Modality*, Artificial Intelligence, vol. 34, (1988), pages 179–212.
64. E. Post, *Formal reductions of the general combinatorial decision problem*, American Journal of Mathematics, vol. 65, (1943), pages 197–214.
65. D. Pym, *Proofs, Search and Computation in a General Logic (Thesis: ECS-LFCS-90-125)*, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh, (1990).
66. W. C. Sellar, Ægrot: Oxon: and R. J. Yeatman, Failed M.A. etc. Oxon:, "1066 and All That: a memorable history of England: comprising all the parts you can remember including one hundred and three good things, five bad kings, and two genuine dates," Methuen, London, (1931).
67. N. Shankar, *A Mechanical Proof of the Church-Rosser Theorem*, Journal of the Association for Computing Machinery, vol. 35, (1988), pages 475–522.
68. J. R. Shoenfield, "Mathematical Logic," Addison-Wesley, Reading, Massachusetts, (1967).
69. A. Simpson, *On The Conservativity of Certain Extensions of Arithmetic*, Unpublished, University of Edinburgh, (10 June, 1991).
70. B. Smith, *Reflection and Semantics in a Procedural Language (MIT/LCS/TR-272)*, Laboratory for Computer Science, Massachusetts Institute of Technology, (1982).
71. B. Smith, *Reflection and Semantics in Lisp (TR ISL-5)*, Intelligent Systems Laboratory, Xerox PARC, Palo Alto, California, (1983).
72. C. Smoryński, "Self-Reference and Modal Logic," Springer, New York, (1985).
73. R. M. Smullyan, "Theory of Formal Systems," Princeton University Press, Princeton, New Jersey, (1961).
74. R. Solovay, *Provability interpretations of modal logic*, Israel Journal of Mathematics, vol. 25, (1976), pages 287–304.
75. A. Tarski, *Pojęcie prawdy w jętzkach nauk dedukcyjnch*, Warsaw, (1933).
76. P. Taylor, *Using Constructions as a Metalanguage (ECS-LFCS-88-70)*, Laboratory for The Foundations of Computer Science, Department of Computer Science, University of Edinburgh, (1988).
77. A. Troelstra, "Metamathematical Investigation of Intuitionistic Arithmetic and Analysis (Lecture Notes in Mathematics No. 344)," Springer, Berlin, (1973).
78. A. Turing, *Systems of logic based on ordinals*, Proceedings of the London Mathematical Society, vol. 45, (1939), pages 161–228.
79. R. Weyhrauch, *Prolegomena to a Theory of Mechanised Formal Reasoning*, Artificial Intelligence, vol. 13, (1980), pages 133–170.
80. A. Whitehead and B. Russell, "Principia Mathematica (second edition)," Cambridge University Press, Cambridge, (1925–27).
81. J. McCarthy et. al., "Lisp 1.5 Programmers Manual (second edition)," M.I.T. Press, Cambridge, Massachusetts, (1965).
82. M. Sato, *Theory of Symbolic Expressions (I)*, Theoretical Computer Science, vol. 22, (1983), pages 19–55.

Table of Contents

Habe nun, ach! Philoſophie,
Juriſterei und Medizin,
Und leider auch Theologie!
Durchaus ſtudiert, mit heißem Bemühn.
Da ſteh ich nun, ich armer Tor!
Und bin ſo klug als wie zuvor